An Automatic Representation Optimization and Model Selection Framework for Machine Learning

Von der Fakultät für Ingenieurwissenschaften,
Abteilung Informatik und Angewandte Kognitionswissenschaft
der Universität Duisburg-Essen
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von

Fabian Bürger
aus
Schwelm

1. Gutachter: Prof. Dr. Josef Pauli
2. Gutachter: Jun.-Prof. Dr. Tobias Glasmachers
Tag der mündlichen Prüfung: 7. Juni 2016

Acknowledgment

A PhD thesis requires years of hard work, consisting of reading, programming, evaluating, improving, evaluating again and – finally – months of writing. This cannot be successfully completed without the support of numerous people:

First of all, I would like to express my deep gratitude to my supervisor Prof. Dr. Josef Pauli of the Intelligent Systems Group at the University of Duisburg-Essen. With his advice and expertise he has significantly contributed to the success of this work. Moreover, he had established great working conditions for research assistants like me – ensuring a good but demanding mixture of guidance and scientific freedom. Furthermore, I also want to thank my second referee Jun.-Prof. Dr. Tobias Glasmachers for helpful discussions and his interest in my work.

The successful management of everyday work at the university would not have been possible without friendly colleagues. In particular, I would like to mention Jens Hoefinghoff and Michael Korn who are not only my friends but have also contributed to my work by valuable discussions and ideas. I also appreciate the time with Christoph Buck and Matthias Thurau who have been good partners and friends in our research project at the university. Furthermore, Marion Handke has always been a great help regarding organizational issues of any kind. Leonid Lorenz has helped me with his technical support in numerous cases.

I would like to thank Elena Zimmermann for proofreading my thesis, which has significantly contributed to the linguistic quality of this work. Furthermore, I was glad to be able to obtain valuable real-world data for my experiments from a local steel factory and I would like to especially thank Thomas Schlüter and Alexey Nagaytsev for their cooperation and discussions.

I also owe a special thank you to Jörn Malzahn for being my adviser of my diploma thesis at the University of Dortmund. The time I spent there has contributed to my interest in becoming a researcher.

My parents, Edith and Stephan Bürger, play a special role in my life and have always supported me in every possible way to achieve my goals. This work, along with countless other things, would not have been possible without their dedication and trust in me. The same applies to my brother Christoph

Bürger. I also like to thank my girlfriend Flore Decharnia to accompany and support me on my way with her love and sympathy.

Last but not least, I want to thank my old and new friends and colleagues for an unforgettable time during the last years!

Fabian Bürger

Abstract

The classification problem is an important part of machine learning and occurs in many application fields like image-based object recognition or industrial quality inspection. In the ideal case, only a training dataset consisting of feature data and true class labels has to be obtained to learn the connection between features and class labels. This connection is represented by a so-called classifier model. However, even today the development of a well-performing classifier for a given task is difficult and requires a lot of expertise. Numerous challenges occur in real-world classification problems that can degrade the generalization performance. Typical challenges are not enough training samples, noisy feature data as well as suboptimal choices of algorithms or hyperparameters.

Many solutions exist to tackle these challenges, such as automatic feature and model selection algorithms, hyperparameter tuning or data preprocessing methods. Furthermore, representation learning, which is connected to the recently evolving field of deep learning, is also a promising approach that aims at automatically learning more useful features out of low-level data. Due to the lack of a holistic framework that considers all of these aspects, this work proposes the *Automatic Representation Optimization and Model Selection Framework*, abbreviated as AROMS-Framework. The central classification pipeline contains feature selection and portfolios of preprocessing, representation learning and classification methods. An optimization algorithm based on Evolutionary Algorithms is developed to automatically adapt the pipeline configuration to a given learning task. Additionally, two kinds of extended analyses are proposed that exploit the optimization trajectory. The first one aims at a better understanding of the complex interplay of the pipeline components using a suitable visualization technique. The second one is a multi-pipeline classifier with the purpose to improve the generalization performance by fusing the decisions of several classification pipelines.

Finally, suitable experiments are conducted to evaluate all aspects of the proposed framework regarding its generalization performance, optimization runtime and classification speed. The goal is to show benefits and limitations of the framework when a large variety of datasets from different real-world applications is considered.

Contents

Chapter 1

Introduction

Computers have revolutionized many application fields – science, industry and also our personal life. Today they have become ubiquitous in form of smart devices such as smartphones or tablet computers that assist us and deliver almost any information within seconds. Computers have been designed as machines that follow programmed instructions for specific tasks, like solving mathematical equations or store and provide information. In the beginning, the tasks have been very static and all logic had to be implemented by the programmers of the system.

Many tasks in real-world applications can still only be done by humans because the problems are too complex for machines with simple and static programs. Humans have a highly developed intelligence to find reasonable solutions for complex problems. Additionally, we have extraordinary perception and pattern recognition capabilities that help us to analyze and understand our environment. A long-term objective would be to develop machines with similar intelligence and perception abilities, but it is a far way – or maybe even impossible – to create such an artificial general purpose system. However, intelligent systems for specific problems have already been introduced that can aid humans solving tasks or act as fully autonomous systems, e.g., automatic industrial quality inspection, traffic sign recognition in driver assistance systems and medical diagnosis systems.

Intelligent systems require the perception and analysis of the current environment or situation using sensors. The goal is to find a function, also called *model*[1], that explains the connection between input – the perceptions, measurements or derived features of them – and the desired outputs, e.g., class labels or specific values. The challenge is that this connection is usually not

[1]The term "model" originates from the field of statistics in which a suitable probability distribution and its parameters – the *model* – for given samples have to be found. This term was conveyed to the field of machine learning and is widely used for functions that describe the feature distribution, e.g., classifier functions.

trivial and for many quite simple tasks it is already hard or even impossible to find an explicit model.

Machine learning is a versatile and efficient approach to find models for these problems. The idea is that algorithms learn the connection between inputs and desired outputs using training data. One important field of machine learning is the supervised classification task, which is the focus of this work. Humans have the ability to learn and recognize things by just seeing a few examples of a new object category. In an ideal case, machine learning works the same way: A reasonable amount of samples with known input and output values has to be collected which is used to train the classifier algorithm. The goal is to achieve a good generalization in the sense that not only the trained data should be recognized correctly, but also previously unseen instances. Many powerful classifier concepts such as support vector machines or random forests[2] have been introduced that perform already well on a wide range of tasks.

However, in many real-world applications the desired performance is not achieved in such an ideal and fast way. Numerous problems are responsible for standard approaches to fail, like too high-dimensional and noisy feature data, too few training samples, unsuitable classifiers or hyperparameters[3]. An important challenge is the fact that no "best" general purpose machine learning algorithm will ever exist, which is stated by the *no-free-lunch theorem*.

The machine learning community has developed and established numerous approaches to solve single aspects of these challenges. Feature selection is used to select only relevant dimensions and to remove noisy features. Feature preprocessing methods, e.g., rescaling the feature values to a range of $[0, 1] \subset \mathbb{R}$, are used to improve the numerical stability of the learning algorithms. Furthermore, the automatic selection of algorithms and hyperparameters is commonly considered using optimization frameworks.

The properties of the *feature representation* of the input data plays an essential role for the success of any machine learning approach. Sensors usually collect several numeric values for each instance, e.g., digital cameras generate many pixel values. This "raw" data is often considered as a low-level representation because it is noisy and shows a high variation even for instances of the same class. This low-level data is normally not well suited to be directly used for machine learning. A lot of manual effort, task-specific expertise and also creativity is necessary to develop suitable high-level features out of the raw data to achieve a good performance – this makes the development of such systems difficult and expensive in practice.

An emerging field of recent research is *representation learning* which tries to find general learning principles for the automatic generation of better suitable features:

[2]These and more classifier concepts are described in chapter 2.1.

[3]A hyperparameter controls the behavior of a learning algorithm itself, see chapter 2.2.

> "To expand the scope and ease of applicability of machine learning, it would be highly desirable to make learning algorithms less dependent on feature engineering so that novel applications could be constructed faster, and more importantly, to make progress toward artificial intelligence (AI). An AI must fundamentally understand the world around us, and we argue that this can only be achieved if it can learn to identify and disentangle the underlying explanatory factors hidden in the observed milieu of low-level sensory data." [Bengio et al., 2013]

The aspect of automatic generation of better representations is a key factor in the recently evolving field of *deep learning* which makes use of artificial neural networks consisting of multiple, complex processing layers. The training algorithms of such networks often utilize vast amounts of training data, e.g., from the internet. Deep learning has already led to a great boost in performance in image classification and speech recognition applications.

Automatic feature construction is a promising approach to generate better feature representations. This field comprises so-called *manifold learning* and *feature transforms* for dimensionality reduction. These methods have basically the same goal to automatically learn a feature transform function from the input data that maps the low-level features into higher-level features. The new representation may be better suitable for simpler or even linear classifiers and helps to improve the performance. Research in neuroscience shows evidence that a probably similar process is going on inside the human or mammalian brain when difficult tasks such as visual object recognition are performed. A great number of automatic feature construction algorithms with different approaches has been introduced, e.g., the well known Principal Component Analysis (PCA), Isomap, Local Linear Embedding (LLE) or deep neural networks such as Autoencoders, just to name a few[4].

Altogether, there is a large portfolio of established approaches as well as representation learning methods that all aim at improving the performance of machine learning. A manual selection of the optimal feature subset, algorithm and hyperparameter combination is nearly infeasible which motivates the use of automatic optimization frameworks. There are numerous of such frameworks available that, e.g., select features and optimize hyperparameters. However, the promising aspect of representation learning is rarely considered yet. Therefore, the central goal of this work is the development of a holistic optimization framework that includes all aforementioned components with a special focus on representation learning by means of feature transforms and manifold learning techniques.

[4]These concepts of automatic feature construction methods are described and referenced in chapter 3.3 and the appendix C.

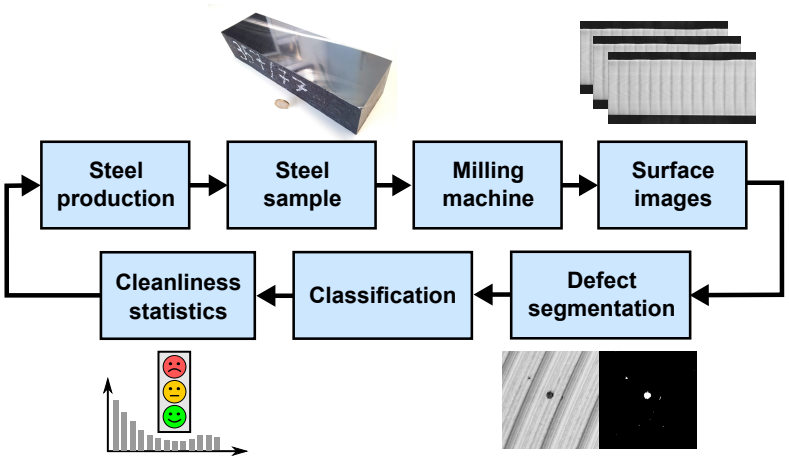
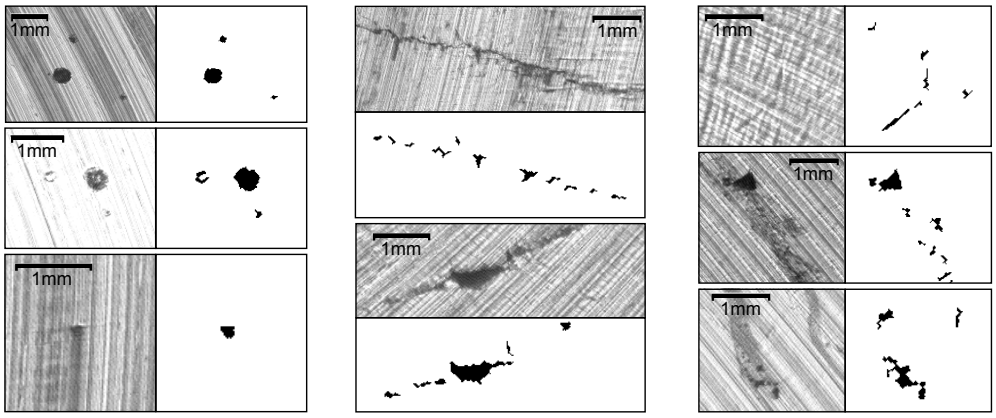


Figure 1.1: Overview of the processing of an image-based measurement system for the cleanliness of steel. The results of the cleanliness statistics are used to improve the production process.

The introduction chapter is organized as follows. In section 1.1 an example classification application is presented to discuss the main challenges that occur in many machine learning problems. These challenges motivate a holistic solution presented in this work. Section 1.2 lists central goals and contribution of this work. Finally, section 1.3 describes the organization and connections of the following chapters.

1.1 Applications and Motivation

Real-world classification problems bear a number of challenges that make it hard to find a classification algorithm with the desired performance. An example application for a difficult classification task can be found in a measurement system for the cleanliness of steel samples presented in [Bürger et al., 2013] and [Buck et al., 2013]. The cleanliness of steel is a crucial parameter for its applications and is defined by the amount, size, chemical composition and distribution of non-metallic inclusions. The measurement system should provide a fast estimation of the cleanliness with the help of image-based analysis of milled steel samples. Figure 1.1 shows the overview of the measurement process. A steel sample is sliced into thin layers with a milling machine and each surface layer is imaged with a camera sensor so that a stack of high-resolution surface images ($10 - 20\mu m$ per pixel) is obtained. These images show the steel surface with a regular milling pattern while defects appear as irregularities. These defects are located and segmented using an irregularity detector for textures [Bürger and Pauli, 2013, Herwig et al., 2013].



(a) Inclusion candidates (b) Cracks (c) Artifacts

Figure 1.2: Exemplary images and corresponding defect segmentation of the object classes inclusions, cracks and artifacts from the steel cleanliness measurement system of [Bürger et al., 2013]. The detected object areas are denoted in black while the background is white. Note that the segmentation method is optimized for the inclusion candidates and therefore all other objects, especially thin cracks, may sometimes be only partially segmented.

There are basically three types of defects that are detected using this method, which are depicted in figure 1.2. The most important class contains *inclusion candidates*, which are shown in figure 1.2 (a). They can be either solid inclusions or gas-filled pores. The second class contains different types of cracks (see figure 1.2 (b)) that form during the casting and solidification process. Finally, the last class contains undesired artifacts that are introduced by the measurement process itself (see figure 1.2 (c)) such as irregular milling grooves or oil drops.

These three classes have to be distinguished in order to achieve a reliable estimation of the steel cleanliness. Human experts are able to distinguish these classes in most cases when they see the image regions. This indicates that the image data contains enough information so that an automated classification is theoretically possible.

Three steps have to be made for the development of an image-based object recognition approach. At first, a set of texture and shape features (see appendix A) needs to be derived from the objects that especially describe the differences between the classes. Secondly, a reasonable amount of ground truth data in form of annotated objects has to be obtained. Finally, a suitable classifier has to be trained with the extracted feature data and the ground truth labels from the training dataset.

This application is a good example for a difficult classification task because of the following reasons:

- The input data is expected to be suboptimal since the objects are relatively small compared to the sensor resolution and the texture is expected to be noisy. It is not obvious which feature set should be chosen.
- There is a low interclass variance because the classes are hard to distinguish and a high intraclass variance as the objects of the same class may look fairly different. The latter is especially the case for the classes cracks and artifacts.
- There are only relatively few training samples available as the generation of ground truth training data is expensive due to a manual inspection of each object with a microscope.
- No former experience with respect to the choice of classifiers or hyperparameters is available for this application.

Even experts in the field of machine learning will not be able to develop such an image-based classification system in a straightforward way. It will likely be a time-consuming and expensive trial-and-error process to reach the necessary high performance and reliability. However, it cannot even be guaranteed that the performance requirements can be achieved at all. In this case, the whole machine learning approach would become infeasible and an alternative measurement system will have to be used – while a lot of valuable development time is wasted.

1.2 Goals and Contributions

The concept of this work is summarized in figure 1.3, including challenges, approaches and goals. First of all, the relevant challenges of typical machine learning tasks are analyzed and discussed. The central contribution is the development of a *holistic framework* that incorporates a large set of solutions for those challenges. This set contains feature selection, feature data representation improvement, classifier selection and hyperparameter selection. The involvement of representation improvement in form of automatic feature construction and manifold learning methods is particularly novel within such a holistic framework.

The main goal of the framework is a *high classification performance* compared to standard classifiers and other state-of-the-art approaches. In order to achieve this, all the components and solutions of the framework need to be tuned for each learning task, which can be considered as a highly combinatorial optimization problem. The properties of this problem are analyzed and a suitable optimization algorithm is developed that should work *fully automatic* and reasonably *fast*. Only relatively *little expertise* should be required to apply the framework successfully on a wide range of learning tasks. It is also wishful that *useful statistics* are generated that allow a deeper insight into

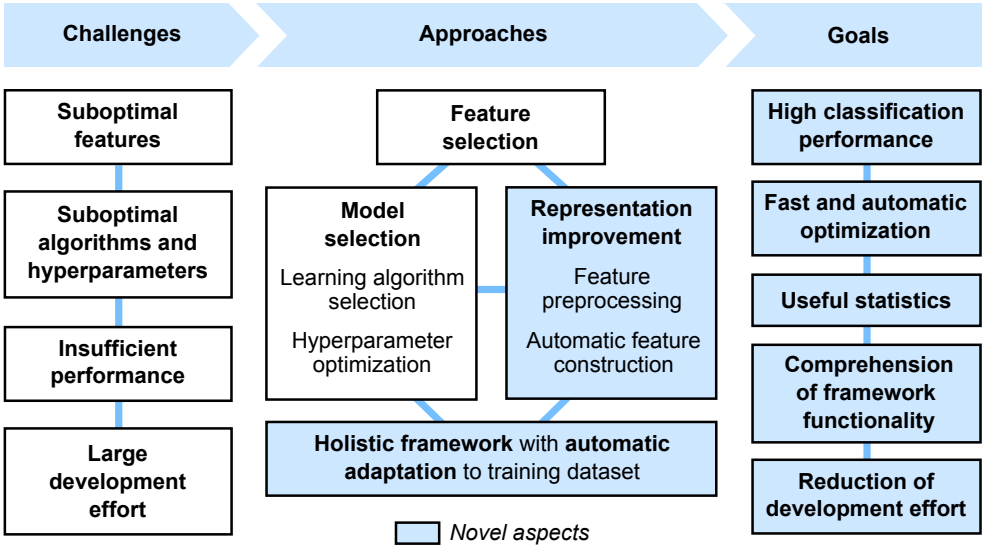


Figure 1.3: Concept overview of this work with challenges, approaches and goals. The highlighted boxes denote areas with novel aspects compared to previous work.

the optimization process and the *framework functionality*. Finally, all these goals are supposed to contribute to the *reduction of the overall development effort* for machine learning systems.

The expectations towards such a holistic framework may be great because the application fields are almost unlimited. However, it should be evident that the goal of a general purpose machine learning system with a human-like recognition performance is *not* realistic – such a system will likely never exist.

1.3 Organization of Chapters

The chapters of this work are organized as depicted in figure 1.4. Chapter 2 presents necessary foundations about machine learning and the supervised classification problem as well as central challenges of these approaches in practical applications. Three kinds of solutions for the challenges are discussed in chapter 3, namely optimization heuristics, established machine learning solutions and representation learning. The great amount of challenges and solutions motivate a holistic framework which is the main contribution of this work.

The *Automatic Representation Optimization and Model Selection Framework* (AROMS-Framework) is presented in chapters 4, 5 and 6. The general structure of the proposed framework and the central classification pipeline are described in chapter 4. This pipeline can be highly adapted to each clas-

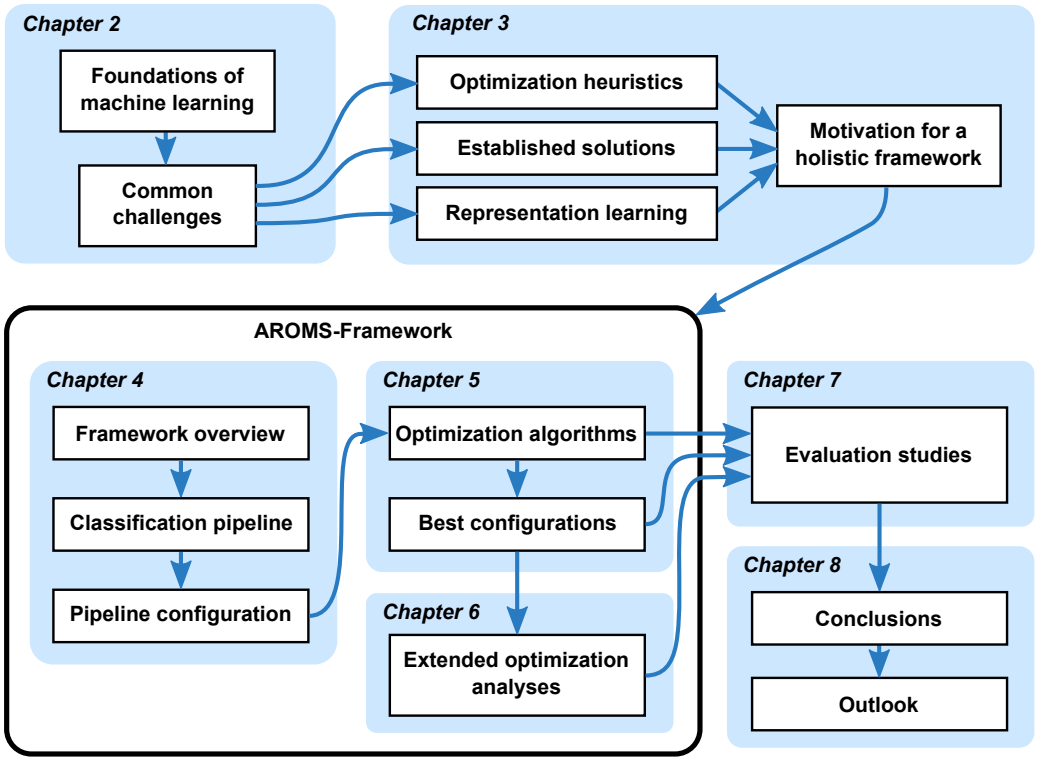


Figure 1.4: Diagram of the organization of chapters and the links between them.

sification problem using the pipeline configuration. This adaptation process is basically a complex optimization problem which is discussed in chapter 5. First, the complexity of the problem is analyzed and a suitable objective function is defined. Then an optimization algorithm is proposed to find solutions for this highly combinatorial problem within a reasonable amount of time. The last part of the framework contains extended optimization analyses which are presented in chapter 6. These analyses exploit the optimization trajectory for two purposes. First, the statistical analysis of the distribution of the best performing components allows an insight into the classification problem and important factors. Visualization techniques can provide a fast overview and understanding of the solution distribution. Secondly, the set of the best pipeline configurations can be used to build multi-pipeline classifiers that have the potential to significantly improve the generalization performance.

Chapter 7 presents the evaluation of the proposed AROMS-Framework with all its relevant parts. The object recognition task of the aforementioned steel cleanliness system as well as other datasets are considered. Finally, chapter 8 summarizes the work and gives ideas for future improvements and potentials based on the contributions of this work.

Chapter 2

Foundations and Challenges in Machine Learning

Machine learning describes a broad family of algorithms that basically learn concepts from data. It has applications in classification, approximation, regression and clustering problems. This work focuses on the supervised classification problem, however, many of the contents also apply for the general field of machine learning. It is important to understand the foundations of machine learning and its typical challenges in order to be able to design practically useful systems.

This chapter is organized as follows. First, the basic definitions of the supervised classification problem as well as an overview of popular classifier concepts are presented in section 2.1. Section 2.2 lists central problems and challenges of machine learning and specifically classifiers that often lead to a bad performance of such systems on real-world datasets. Finally, in section 2.3 a discussion about chances and challenges of machine learning concludes this chapter.

2.1 The Supervised Classification Problem

The supervised classification problem can be found in many important, real-world applications such as image-based object recognition or medical diagnosis systems. It is also referred to as pattern recognition problem and many textbooks like [Bishop, 2006] and papers such as [Jain et al., 2000] provide extensive overviews of this topic.

Machine learning methods can generally be subdivided into *supervised*, *unsupervised* and *semi-supervised* methods. The "classical" supervised classification problem can be defined in the following way. A feature vector is defined as

$$\mathbf{x} = [x_1, x_2, \ldots, x_{D_{in}}] \in \mathbb{R}^{D_{in}} \tag{2.1}$$

containing D_{in} scalar components. If feature vectors appear in a list or *dataset*, they are usually denoted with an index $\mathbf{x}^{(i)}$. In this case the jth scalar component of the ith feature vector is written as $x_j^{(i)}$. For classification a feature vector needs to be assigned to a class label y out of a discrete set of $N_{Classes}$ classes or categories $y \in S_{Classes} = \{\omega_1, \omega_2, \ldots, \omega_{N_{Classes}}\}$. The goal is to find a classifier function

$$f_{Classifier} : \mathbf{x} \rightarrow y \in S_{Classes}\,, \tag{2.2}$$

which is often considered as the classifier *model*. This model is adapted to the learning task by training data in form of a ground truth dataset T_{GT} of $1 \leq i \leq N_T$ labeled feature vectors

$$T_{GT} = \left\{ \left(\mathbf{x}^{(1)}, y^{(1)}\right), \left(\mathbf{x}^{(2)}, y^{(2)}\right), \ldots, \left(\mathbf{x}^{(N_T)}, y^{(N_T)}\right) \right\} \tag{2.3}$$

with $y^{(i)} \in S_{Classes}$.

In contrast to that, *unsupervised* learning methods just rely on the input vectors $\{\mathbf{x}^{(i)}\}$ and do not require the corresponding labels $y^{(i)}$. A typical application for unsupervised methods is *clustering*: The goal is to discover natural groupings or accumulations inside the data distribution [Jain et al., 2000]. *Semi-supervised learning* methods make use of both unlabeled and labeled feature data to train a classifier, which can be beneficial when it is too expensive to obtain class labels for all training instances.

2.1.1 Types of Parameters in Machine Learning

Three different kinds of *parameters* are used in the field of machine learning and optimization which have to be clearly differentiated. These parameters as well as their hierarchy are defined in the following way within this work:

1. *Model parameters* are used by the classifier model inside of the decision function $f_{Classifier}$. These model parameters are responsible to store the concepts derived from the training dataset. Examples of these parameters are network weights in case of artificial neural networks or support vectors in case of the support vector machine (see section 2.1.2). The model parameters are on the lowest level of the parameter hierarchy as they directly influence the classifier. These parameters are also known as *internal parameters* or *variables*.
2. *Hyperparameters* of learning algorithms control the way in which the classifier model and its model parameters are adapted or trained to the ground truth dataset. Examples for hyperparameters are the number of layers in artificial neural networks or the regularization hyperparameter of support vector machines (see section 2.1.2). The term "hyperparameter" is very established in literature and is considered to be in the middle of the parameter hierarchy.

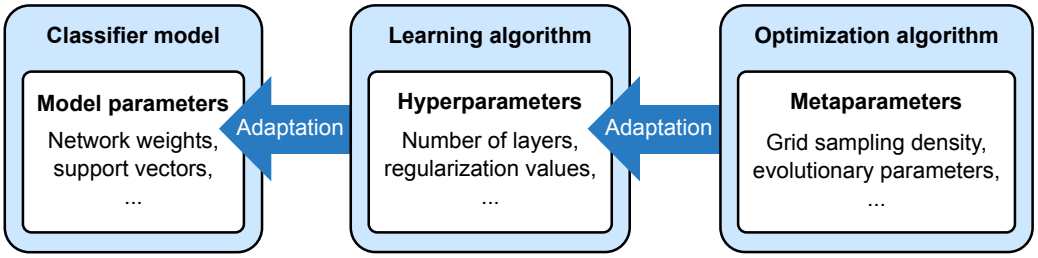


Figure 2.1: Connection between model parameters, hyperparameters and metaparameters in machine learning.

3. *Metaparameters* are on the highest[1] level of the parameter hierarchy and control optimization algorithms to adapt learning algorithms. The hyperparameters of learning algorithms have a great effect on the classification performance and need to be tuned for each learning task (see section 2.2.4). Examples of metaparameters are the grid sampling density in case of grid search or the parameters of Evolutionary Algorithms (see section 3.1.1). Metaparameters are also known as *control parameters*, *behavioral parameters* or sometimes even referred to as "*hyper-hyperparameters*".

Figure 2.1 shows the location of these three types of machine learning parameters as well as the connections between them. Note that the use of optimization algorithms for machine learning is not considered as absolutely necessary but very helpful to improve the performance (see section 3.1.1).

2.1.2 Classifier Concepts

There is a great number of different approaches to model a classifier function $f_{Classifier}$ and it is a highly active field of research. This section just gives a coarse overview of currently relevant classifier concepts, their principles and important hyperparameters.

According to [Kotsiantis, 2007] classifiers can be subdivided into five categories: logic-based algorithms, artificial neural networks, statistical learning algorithms, instance-based learning and support vector machines. These concepts and some of the most popular classifiers are described in the following.

Logic-Based Algorithms

Logic-based algorithms learn formal decision rules from observations that should explain the class distribution of the training data. One form of these

[1]The prefixes "meta" and "hyper" do not naturally imply a hierarchy. However, the metaparameters are chosen to be one level up in the parameter hierarchy as the term hyperparameter is already established in literature.

algorithms are *rule-based learning* methods that automatically build Boolean rules in disjunctive normal form. In the period between 1969 and 1996 a large variety of these algorithms has been developed and the most popular variants are reviewed in [Fürnkranz, 1999]. The training is often performed using divide-and-conquer strategies. However, pure rule-based learning algorithms have become less popular nowadays and are outperformed by other learning methods.

The concept of *decision trees* is related to rule-based learners, but they build a graph out of hierarchical structures to recursively split up the feature data space. For numerical values tree nodes contain simple split rules like

$$\text{if } x_3 > 4 \text{ then } a \text{ else } b \tag{2.4}$$

in which x_3 is the third feature vector component and a, b are child nodes of the current one. Classifiers based on decision trees have several advantages: They require only few assumptions about the data distributions and so they work on a large range of tasks. They have a built-in feature selection of relevant features. Furthermore, the classification speed is often high because only simple rules need to be evaluated. An overview of decision tree based classifiers can be found in [Murthy, 1998]. The training algorithms have to find the optimal variables and split values for the tree nodes to construct the graph. This is often achieved using random selection and information theoretic metrics like information gain based on entropy. Important hyperparameters of random trees are the maximum depth of the tree which determines the complexity of the decision function.

Several trees can be grouped together by means of so-called ensemble methods. A popular concept is the *random forest* (RF) classifier introduced by [Breiman, 2001], which fuses together multiple decision trees using a bagging algorithm. Bagging is a method to generate multiple classifiers or predictors that are trained from differently subsampled training datasets [Breiman, 1996] (see also section 6.3.1). The random component of this method ensures that the trees are less correlated to each other to improve their predictive performance. A central hyperparameter of random forests is the number of trees that should be fused. When the number of trees is sufficient, very complex and high-dimensional feature distributions can be handled with random forests.

Artificial Neural Networks

Artificial neural networks denote data structures and algorithms that are inspired by the functioning of the brain of animals and humans or other parts of the nervous system. The *perceptron*, developed by [Rosenblatt, 1962], is the most important basic model of a single neuron which is still used in currently relevant algorithms. It has D_{in} input values for each vector component of

$\mathbf{x} \in \mathbb{R}^{D_{in}}$ and one binary output value $y \in \{0, 1\}$. The memory of such a neuron is realized in a weight vector $\mathbf{w} \in \mathbb{R}^{D_{in}}$ which contains a weight for each input. The processing is performed with the linear associator as propagation function, which is basically a dot product of the input vector with the weight vector. If the dot product exceeds a threshold – also called bias – b, the neuron's binary output is activated with

$$y = \begin{cases} 1 & \text{if } \sum_{j=1}^{D_{in}} w_j \cdot x_j > b \\ 0 & \text{else} \end{cases}. \tag{2.5}$$

A single neuron is quite limited and only allows solving linear[2] classification problems. Many variants of neural networks have been developed to overcome this limitation.

The connection of multiple neurons on multiple layers as a feedforward network is known as *multilayer perceptron* (MLP) and allows non-linear classification. The input layer of the MLP contains D_{in} nodes to connect with the input vectors. The output layer has a task-specific number of neurons and the outputs often represent the probabilities of the specific classes. An important hyperparameter of an MLP is the number of hidden layers between the input and output layer as well as the number of neurons on each of these layers. Each neuron has weights that have to be trained so that the whole network leads to the desired outputs. One popular training algorithm is *backpropagation* [Bishop, 2006] in which the training vectors of each instance are passed through the network. The error between the current and the desired output is used to adapt the weights using gradient descent methods. Backpropagation methods are relatively slow for large networks and tend to overfit to the training dataset.

Another variant is the so-called *Extreme Learning Machine* (ELM) which has been introduced by [Huang et al., 2006]. It is a feedforward neural network with one hidden layer in which the weights on the hidden layer are chosen randomly and independently from the training dataset. The training process analytically determines the weights on the output layer using least-squares fitting. This makes the training process much faster than the MLP while the ELM also shows a good generalization performance for many tasks.

Statistical Learning Algorithms

Statistical methods build an underlying probability model to describe the class distribution in the feature space. One example is the *naïve Bayes classifier* that uses the strong assumption of independence of all features [Bishop, 2006]. The probability distribution of each feature is modeled with a suitable probability distribution, e.g., a Gaussian normal distribution, whose model

[2]In this simple case, the instances are completely separable using a linear decision boundary or hyperplane.

parameters are estimated during the training. The classification can be realized efficiently by calculating a product function over all features for each class and choosing the class with the highest likelihood. Despite the simplicity of the model and the fact that no further hyperparameters are required, the naïve Bayes classifier shows a reasonable performance for high-dimensional problems such as text classification.

The statistical concept of *finite mixture models* is another approach for classification. An extensive overview can be found in [McLachlan and Peel, 2000]. The basic idea is to model a complex probability distribution of the classes in the feature space with a number of simple distributions, e.g., Gaussian distributions, which are called components. The mixed likelihood distribution of a class is obtained by the superposition of all components. Finally, the maximum mixed probability along all classes is used to assign a label to an instance vector $\mathbf{x}$. In order to learn such a model, the number of components has to be determined and then their model parameters have to be fitted to the data distribution using expectation maximization.

Instance-based Learning

Instance-based classifiers belong to the group of algorithms that store the entire training dataset and only use specific instances for the classification [Aha et al., 1991]. This is also called *lazy learning* because the generalization is delayed to the moment when an instance has to be classified [Aha, 1997]. One particularly known example is the nearest neighbor algorithm which was introduced by [Cover and Hart, 1967]. A new instance $\mathbf{x}$ is classified in the following way. The N_{Neigh} nearest neighbors[3] in the training dataset to the vector $\mathbf{x}$ are searched and the most frequent label of the selected training instances is returned. One important hyperparameter of the kNN classifier is the distance metric $d(\mathbf{x}^{(a)}, \mathbf{x}^{(b)})$ between two points $\mathbf{x}^{(a)}$ and $\mathbf{x}^{(b)}$ in $\mathbb{R}^{D_{in}}$. According to [Kotsiantis, 2007] several metrics are useful in different applications, e.g.,

- the Euclidean distance with $d_{Euclid}(\mathbf{x}^{(a)}, \mathbf{x}^{(b)}) = \sqrt{\sum_{j=1}^{D_{in}} \left(x_j^{(a)} - x_j^{(b)}\right)^2}$,
- the Mahalanobis distance with $d_{Mahal}(\mathbf{x}^{(a)}, \mathbf{x}^{(b)}) = \sqrt{\left(x_j^{(a)} - x_j^{(b)}\right)^\top \Sigma^{-1} \left(x_j^{(a)} - x_j^{(b)}\right)}$ with the covariance matrix Σ of the distribution of all vectors,
- the Chebychev distance with $d_{Cheby}(\mathbf{x}^{(a)}, \mathbf{x}^{(b)}) = \max_j \left(\left|x_j^{(a)} - x_j^{(b)}\right|\right)$,
- the Manhattan or city block distance with $d_{Man}(\mathbf{x}^{(a)}, \mathbf{x}^{(b)}) = \sum_{j=1}^{D_{in}} \left|x_j^{(a)} - x_j^{(b)}\right|$.

[3] The number of neighbors is often denoted as k, hence this method is often referred to as kNN classifier.

The second hyperparameter N_{Neigh} controls the smoothness of the decision boundary between the classes. The disadvantage of the kNN classifier is that it has large storage requirements and is sensitive to noise for small values of N_{Neigh}.

Support Vector Machines

The concept of *support vector machines* (SVM), which was formulated by [Cortes and Vapnik, 1995], is very popular and shows a great generalization performance in a wide range of applications. A detailed tutorial about the SVM can be found in, e.g., [Burges, 1998]. For a simple binary classification problem with $y^{(i)} \in \{+1, -1\}$ the main idea is to find an optimal separating hyperplane with a maximum margin between two classes that is equal to a structural risk minimization. The closest training points to this hyperplane are called support vectors. The SVM stores these support vectors, which are usually only a small fraction of the total training dataset. The separating hyperplane is defined by an orthogonal weight vector $\mathbf{w} \in \mathbb{R}^{D_{in}}$ and a bias value $b \in \mathbb{R}$. Once $\mathbf{w}$ and b are found, the classification can easily be done by

$$f(\mathbf{x}) = \mathrm{sgn}(\mathbf{w}^\top \mathbf{x} + b) \tag{2.6}$$

which is very similar to the perceptron. In the simple, linearly separable case the training works the following way. It is defined that

$$\mathbf{w}^\top \cdot \mathbf{x}^{(i)} + b = \begin{cases} \geq 1 \quad , & y^{(i)} = +1 \\ \leq -1 \, , & y^{(i)} = -1 \end{cases} \tag{2.7}$$

which leads to a set of inequalities

$$y^{(i)} \cdot (\mathbf{w}^\top \cdot \mathbf{x}^{(i)} + b) \geq 1 \tag{2.8}$$

for $1 \leq i \leq N_T$. It can be shown that the width of the margin is simply $2/||\mathbf{w}||$ and thus minimizing $||\mathbf{w}||$ maximizes the margin. The hyperplane model parameters can be found by optimizing

$$\arg\min_{\mathbf{w},b} \max_{\boldsymbol{\alpha} \geq 0} \left\{ \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{N_T} \alpha^{(i)} y^{(i)} (\mathbf{w}^\top \cdot \mathbf{x}^{(i)} + b) + \sum_{i=1}^{N_T} \alpha^{(i)} \right\} \tag{2.9}$$

with standard quadratic optimization methods. The non-negative factors $\alpha^{(i)}$ are called Lagrange multipliers that act as a "switch" to make training vectors support vectors when $\alpha^{(i)} > 0$. The weight vector can be expressed using

$$\mathbf{w} = \sum_{i=1}^{N_T} \alpha^{(i)} y^{(i)} \mathbf{x}^{(i)}, \tag{2.10}$$

which clarifies that all vectors with $\alpha^{(i)} = 0$ have no impact on the classification.

In order to use SVMs in non-separable cases the soft margin method has been developed. To achieve this, slack variables $\xi^{(i)} \geq 0$ are introduced in

$$y^{(i)}(\mathbf{w}^\top \cdot \mathbf{x}^{(i)} + b) \geq 1 - \xi^{(i)}. \tag{2.11}$$

The goal is the permission of a certain fraction of misclassifications, which is controlled by a regularization factor that penalizes them in the adapted optimization function

$$\begin{aligned} \arg\min_{\mathbf{w},\xi,b} \max_{\boldsymbol{\alpha},\boldsymbol{\beta}} \Bigg\{ & \frac{1}{2}\|\mathbf{w}\|^2 + c_{reg} \sum_{i=1}^{N_T} \xi^{(i)} \\ & - \sum_{i=1}^{N_T} \alpha^{(i)}[y^{(i)}(\mathbf{w}^\top \cdot \mathbf{x}^{(i)} + b) - 1 + \xi^{(i)}] - \sum_{i=1}^{N_T} \beta^{(i)}\xi^{(i)} \Bigg\} \end{aligned} \tag{2.12}$$

with Lagrange factors $\alpha^{(i)}, \beta^{(i)} \geq 0$. The hyperparameter c_{reg} is important to control the regularization cost of misclassifications and therefore this concept is known as C-SVM.

Additionally, the concept of SVM has been extended to work for non-linear problems using kernel methods. It allows a transformation of a non-linear problem into a higher-dimensional feature space in which the problem is solvable by using a linear hyperplane. The "trick" is that the projection into a very high-dimensional space does not need to be calculated explicitly to save computation time. Instead, a kernel function $K(\mathbf{x}^{(a)}, \mathbf{x}^{(b)})$ replaces the dot product in the decision function

$$f(\mathbf{x}) = \operatorname{sgn}\left(\sum_{i=1}^{N_T} \alpha^{(i)} y^{(i)} K\left(\mathbf{x}, \mathbf{x}^{(i)}\right) + b \right). \tag{2.13}$$

There are a number of popular kernel functions, e.g.,

- the polynomial kernel

$$K\left(\mathbf{x}^{(a)}, \mathbf{x}^{(b)}\right) = \left(\mathbf{x}^{(a)} \cdot \mathbf{x}^{(b)} + \delta\right)^{\eta} \tag{2.14}$$

  in which δ denotes an additive constant and η the degree of the polynomial,

- the Gaussian kernel or radial basis function kernel

$$K\left(\mathbf{x}^{(a)}, \mathbf{x}^{(b)}\right) = \exp\left(-\gamma_{Gauss}\|\mathbf{x}^{(a)} - \mathbf{x}^{(b)}\|^2\right) \tag{2.15}$$

  in which γ_{Gauss} is a kernel hyperparameter that is reciprocal to the width of the Gaussian and controls the smoothness of the decision boundary.

The choice of the kernel function and its hyperparameters have a great impact on the learning behavior and thus need to be adapted for each learning task.

Predicted		*True class* positive	*True class* negative
	positive	true positives (TP)	false positives (FP)
	negative	false negatives (FN)	true negatives (TN)

Table 2.1: Binary confusion matrix for the two class problem.

2.1.3 Performance Evaluation

It is important to evaluate the performance of a classifier on a specific task to see whether or not the desired reliability of the system is achieved. This is usually done by splitting the entire training dataset T_{GT} into a separate training and test dataset

$$T_{GT} = T_{Train} \cup T_{Test} \,, \tag{2.16}$$

which have to be disjoint. Normally, the division ratio of T_{Train} and T_{Test} ranges from 50/50% to 90/10%. The classifier is trained using the training dataset T_{Train} and the performance is measured using the prediction of the classifier on T_{Test} compared to the true labels. Note that more sophisticated methods like cross-validation (see section 3.2.1) even further split up the training dataset T_{Train}.

There are standard metrics to quantify the performance of classifiers which can be found in, e.g., [Japkowicz and Shah, 2011] or [Fawcett, 2006]. Table 2.1 illustrates the binary confusion matrix for the two class problem with $S_{Classes} = \{\omega_1, \omega_2\}$, which is a standard way to count all possible cases of true and false predictions. In this case instances of class ω_1 are seen as "positive", whereas instances of class ω_2 are seen as "negative".

The classifier prediction of an instance can be categorized as either true positive (TP), true negative (TN) or in case of an error as false positive (FP) or false negative (FN). The frequencies of all these four cases are counted over the whole test dataset T_{Test} and are respectively denoted as N_{TP}, N_{TN}, N_{FP} and N_{FN}.

Based on these four numbers, several important metrics can be derived. First, the *true positive rate* (also known as recall rate or hit rate) is defined as

$$q_{TPRate} = \frac{N_{TP}}{N_{TP} + N_{FN}} \tag{2.17}$$

and the *false positive rate* (or false alarm rate) as

$$q_{FPRate} = \frac{N_{FP}}{N_{FP} + N_{TN}} \,. \tag{2.18}$$

The *precision* metric is defined as

$$q_{Prec} = \frac{N_{TP}}{N_{TP} + N_{FP}} \tag{2.19}$$

Predicted \ *True class*	ω_1	ω_2	$\dots$	$\omega_{N_{Classes}}$
$\widetilde{\omega}_1$	$N_{1,1}$	$N_{1,2}$	$\dots$	$N_{1,N_{Classes}}$
$\widetilde{\omega}_2$	$N_{2,1}$	$N_{2,2}$	$\dots$	$N_{2,N_{Classes}}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$\widetilde{\omega}_{N_{Classes}}$	$N_{N_{Classes},1}$	$N_{N_{Classes},2}$	$\dots$	$N_{N_{Classes},N_{Classes}}$

Table 2.2: Multiclass confusion matrix with frequencies N_{k_1,k_2} of all possible correct and erroneous cases.

and denotes the relevance of the results of a classifier with respect to class ω_1. A low precision indicates a high false alarm rate. The *accuracy* is defined as

$$q_{Acc} = \frac{N_{TP} + N_{TN}}{N_{TP} + N_{FN} + N_{FP} + N_{TN}} \tag{2.20}$$

and denotes the fraction of correctly classified instances. The values of q_{Acc} are in the range of $[0, 1] \subset \mathbb{R}$ while a higher value usually denotes a better classifier. If these values are multiplied with the factor 100, they can directly be interpreted as the percentage of correct predictions. The inverse accuracy is equivalent to the *error rate*, which is defined as

$$q_{Err} = 1 - q_{Acc} \,. \tag{2.21}$$

Many real-world problems consider the distinction of more than two classes. This makes it necessary to define useful metrics for multiclass problems as well which can be found in, e.g., [Labatut and Cherifi, 2011] and [Japkowicz and Shah, 2011]. The multiclass confusion matrix is defined as in table 2.2. It contains all possible combinations of true classes denoted as ω_k and predicted classes denoted as $\widetilde{\omega}_k$.

The most important metric is the *overall accuracy* or *overall success rate* that is widely used in literature. It is defined as

$$q_{OAcc} = \frac{1}{N_{all}} \sum_{k=1}^{N_{Classes}} N_{k,k} \quad \text{with} \quad N_{all} = \sum_{k_1=1}^{N_{Classes}} \sum_{k_2=1}^{N_{Classes}} N_{k_1,k_2} \tag{2.22}$$

which is basically the number of correctly classified instances divided by the number of all classifications. The *overall error rate* is analogously defined to the binary case as

$$q_{OErr} = 1 - q_{OAcc} \,. \tag{2.23}$$

The q_{OAcc} and q_{OErr} metrics are global for all classes and can be skewed when the numbers of instances are heavily unequal for the different classes. Consider a test dataset with 10 instances for class ω_1 and 1,000 instances for class ω_2. A classifier might predict the label $\widetilde{\omega}_2$ for all instances regardless

of the input. Due to the class sample imbalance, this classifier achieves an overall accuracy of

$$q_{OAcc} = \frac{0 + 1{,}000}{1{,}000 + 10} \approx 0.99\,. \tag{2.24}$$

This is certainly misleading because this classifier would not be desirable even though its accuracy is around 99%. There are essentially two possibilities to circumvent this problem. The first solution is to obtain a more balanced dataset by either collecting more samples of the classes with too few samples or by removing instances from the classes with too many samples. Secondly, it is possible to derive class-specific metrics from binary confusion matrices for each class. Instances of class ω_k are considered as "positive" and all others ($y \in S_{Classes} \setminus \omega_k$) as "negative". The true positive rate of class ω_1 would be zero for the classifier in the example.

Metrics based on confusion matrices are widely used and can easily be calculated for any classifier. However, more sophisticated methods such as the *receiver operating characteristic* (ROC) have been introduced. The ROC curve analysis is preferred for applications in which the trade-off between the true positive rate and the false positive rate is important [Fawcett, 2006]. The basic ROC curve analysis is defined for a binary classifier which not only returns a class label but also a confidence or score value. The ROC curve is created by sorting the results by their confidence and plot the false positive rate as abscissa axis and the true positive rate as ordinate axis. In order to obtain a single number for comparison, the *area under the ROC curve* (AUC) has been defined. Usually, a better classifier has a higher AUC value. The fact that the classifier confidence is involved in this metric makes it potentially more valuable to evaluate a classifier rather than just using the discrete class output which is used for the confusion matrices. However, not all classifiers provide a confidence score and even if some do, it is hard to compare the scores across different classifier concepts. Furthermore, there is no trivial way to define a useful scalar metric like the AUC for multiclass problems. Some concepts in that direction can be found in [Fawcett, 2006].

2.2 Main Challenges

In real-world applications, numerous challenges and problems occur that influence the performance of machine learning approaches. Even though the focus of this work is the supervised classification problem, many of the challenges that are mentioned in the following subsections also affect other machine learning methods.

2.2.1 High-Dimensional Data

Many applications need to handle high-dimensional but low-level feature data. Some examples are given in the following:

- *Image-based object recognition* tasks use the data from camera sensors to locate and classify objects. Even rather small textures of the objects create high-dimensional feature spaces. A texture with, e.g., 50×50 pixels already leads to a 2,500 dimensional feature space. A common approach is to extract higher level features from the texture, such as the popular Scale-Invariant Feature Transform (SIFT) descriptor [Lowe, 2004] or the Local Binary Patterns (LBP) descriptor [Ojala et al., 1994, Ojala et al., 2002] (see appendix A). However, the dimensionalities remain rather high with 128 for SIFT and 256 for the standard LBP descriptor.
- *Optical spectrometers* analyze the electromagnetic spectrum of emitted light and usually generate high-dimensional measurements consisting of up to several thousand wavebands[4]. These spectrometers are used in many applications, such as remote sensing in geography [Melgani and Bruzzone, 2004], chemical material characterization [Chang, 2003] and bioinformatics [Somorjai et al., 2004].
- *Microarray analysis* is an application in bioinformatics with the goal to analyze the properties of biological materials. One example are DNA[5] microarrays that analyze gene expressions [Simon, 2003]. The dimensionality of this kind of data can exceed 10,000.

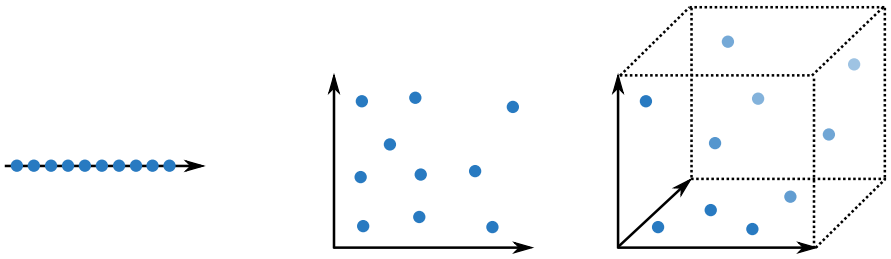
The issues that arise from high-dimensional data have a great effect on the performance of machine learning algorithms. All effects together are often summarized as the well-known *curse of dimensionality*, which was first observed and reported by [Bellman, 1961]. It describes problems that are caused by the interplay of the number of features or dimensions, the number of samples, the feature distribution and the classifier complexity. The curse of dimensionality can be subdivided into three aspects as described in the following.

General Negative Effects of High-Dimensional Spaces

The dimensionality of feature spaces directly influences central properties of them. At first, the amount of training samples or data points that are needed

[4]Optical spectrometers measure the intensity of light in a specific part of the electromagnetic spectrum, such as the visible light in a range of $400 - 800$ nanometers (blue to red colors). The number of wavebands defines the spectral resolution in this range and depends on the specific sensor.

[5]DNA stands for *deoxyribonucleic acid* and denotes molecules that carry genetic information in organisms.



(a) 1D (b) 2D (c) 3D

Figure 2.2: Visualization of the "vastness" of high-dimensional spaces. The figure shows 10 datapoints in a one-, two-, and three-dimensional feature space. While the feature space in (a) is covered densely, the three-dimensional space in (c) appears rather empty.

to cover or "fill" a vector space in a reasonable way grows exponentially with the dimensionality [Bishop, 1995]. Usually, high-dimensional spaces are rather vast and empty – this effect can already be observed in relatively low-dimensional spaces, which is displayed in figure 2.2.

Secondly, canonical distance metrics such as the Euclidean distance become less meaningful in high-dimensional vector spaces. The authors of [Beyer et al., 1999] prove this effect by considering the distance metrics for randomly distributed vectors. It can be shown that the difference between the smallest and the largest distance d_{min} and d_{max} between vectors compared to d_{min} becomes arbitrarily small when

$$\lim_{D_{in}\to\infty} \frac{d_{max} - d_{min}}{d_{min}} = 0\,. \tag{2.25}$$

It means that the smallest and largest distances only differ marginally from each other in high-dimensional vector spaces. This causes the performance degradation of classifiers which rely on these distances.

The third aspect is the so-called *peaking phenomenon* recognized by [Jain and Chandrasekaran, 1982] and [Raudys and Jain, 1991]. It was observed that the error rate of some classifiers tends to first decrease and then increase again with increasing dimensionality D_{in}. When the error is plotted depending on D_{in}, like depicted in figure 2.3, a negative peak – or valley – can be spotted. However, this effect is more complex and other classifiers behave differently as the authors of [Sima and Dougherty, 2008] state. They report three types of classifier behaviors, namely the *peaking type*, the *plateau type* and the *slope type*, which are also depicted in figure 2.3. The peaking phenomenon in particular occurs for classifiers whose number of model parameters depends on the feature dimensionality. When dimensions are added while the sample size stays constant, the classifier has to estimate more model

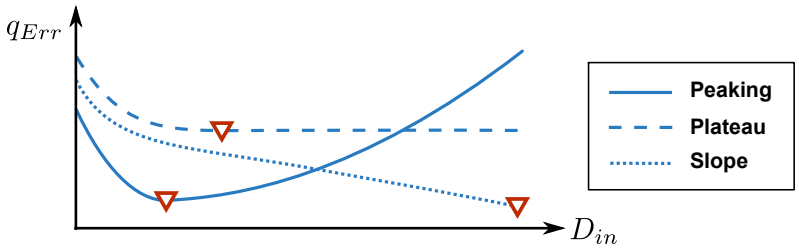


Figure 2.3: Visualization of three different types of classifier behaviors regarding the error q_{Err} depending on the feature dimensionality D_{in}. The lowest error rates are denoted with a triangle. The peaking type has a very small range of optimal numbers of dimensions, hence the name "peaking". The plateau type reaches stable error values when the dimensionality is large enough. The error of the slope type is continuously decreasing with increasing dimensionality.

parameters with the same amount of training samples. Therefore, the reliability of the model parameter estimation process while learning decreases and most likely also the classifier performance. Statistical classifiers that estimate covariance matrices of the feature data are a well studied example for the peaking phenomenon. But, to give another example, also the multilayer perceptron can suffer from this phenomenon because the number of weight variables on the input layer linearly scales with the feature space dimensionality – the higher the dimensionality, the more weights have to be trained with the same number of samples.

Noisy or Irrelevant Features

In many applications there is few or no prior knowledge about a suitable feature set. One strategy is to use a large feature set with many descriptors that appear suitable at first glance. However, noisy, irrelevant or redundant feature dimensions can severely disturb machine learning algorithms. The peaking phenomenon is related to this problem, but does not make an assumption about the quality of the features with respect to the classification task. The work of [Houle et al., 2010] provides evidence that not only the number of dimensions is crucial but the amount of *relevant* data as well. If the feature data contains many dimensions which are not correlated with the desired output at all, the classifier performance can already be degraded for relatively small feature dimensionalities like 20 – 50 dimensions.

Small Training Sample Sizes

The number of training samples directly affects the negative consequences of the curse of dimensionality. Large sets of labeled training data are usually

hard and expensive to obtain so that many applications must rely on rather small training datasets. An important metric is the *sample to feature ratio* (SFR), which is defined as

$$v_{SFR} = \frac{N_T}{D_{in}} \,. \tag{2.26}$$

Different constraints for this metric are proposed, e.g., in [Jain and Chandrasekaran, 1982], a value of $v_{SFR} > 10$ is considered as minimum. However, in many applications the SFR values are far below one. Further discussions on effects of the sample size are presented in [Jain and Zongker, 1997] and [Raudys and Jain, 1991].

It can be summarized that there is no general number of *too many* dimensions. The interplay of all of the aforementioned aspects is of importance. A feature space of 200 dimensions can be perfectly working with one task while it is already too high for another task.

2.2.2 Generalization

Generalization is the ability of systems to learn abstract concepts from observations, also known as induction. Humans are usually very good in understanding abstract concepts, which can be illustrated with an example: When going through a forest, we likely see different kinds of trees, e.g., birch trees, oak trees and firs. These observations can be considered as training samples to understand the abstract concept of trees that might consist of aspects like "object with a vertically elongated, brown trunk, branches and green or brown leaves". In reality this concept is far more complex and hard to explicitly describe. If the generalization process is successful, other kinds of trees, which have not been observed before, e.g., a palm tree, should be recognized as a tree as well.

Bias-Variance Dilemma

Generalization is probably the most desirable property of machine learning systems to come close to the extraordinary capabilities of humans. The main problem of current – and most likely future – machine learning approaches is the bias-variance dilemma [Bishop, 2006]. This can be summarized as follows. There are two sorts of errors, the *bias*, which is the general error of the predicted and the desired output, and the *variance*, which arises from the sensitivity to fluctuations of the data in the training dataset. The theorem states that it is generally impossible to minimize both, the bias and the variance, simultaneously. In practice, it can be observed that an algorithm which is too simple (e.g., a linear classifier) leads to a rather large error, but it produces relatively stable outputs for unseen data. On the other hand, complex classifiers with a great plasticity tend to memorize and basically just

store the training instances – this is known as *overfitting*. Considering the tree example, a too simple classifier would most likely be able to detect a certain fraction of objects as trees correctly, but sometimes fail even for known trees. An overfitted classifier would recognize the trained trees perfectly but most likely fail to classify any other object correctly.

The problem of generalization does not only depend on the classifier's abilities but also on the *invariance* of the feature data representation. An invariant feature should only be marginally affected by any expected transformation or change of the instances that should be classified. In image-based object recognition, e.g., it is often desired that features are invariant to translation, rotation, scaling or illumination to be able to detect objects even if they appear in a different pose or other light conditions. The SIFT descriptor, e.g., implements this invariance and shows a great performance for many recognition applications. However, there is no general purpose feature descriptor and the development of task-specific features with the desired invariance is usually difficult and time-consuming.

Another option to achieve a better feature invariance is a strict control of the conditions under which the feature data is generated. In case of image processing it would be possible to make sure that all objects are aligned in a specific way and an optimized light system is used. Nevertheless, this is only possible to a certain degree and even impossible for some applications. Furthermore, too many restrictions usually limit the application field of the system.

Representational Bias

The aspect of the so-called *representational bias*[6] [Gordon and Desjardins, 1995] also influences the generalization performance. A specific feature set – the feature representation – is usually selected based on the available training data to optimize a certain performance metric such as the accuracy. However, the selected representation might not be optimal to classify unseen instances, which the following simple example shows. Consider a classification problem to distinguish cars from buses based on two features, namely the height and the color. There are two samples in the training set: a blue car with a height of 1,60m and a red bus with a height of 2,20m. A classifier that only knows about the small training set might select only the color feature because it appears to be well suitable to distinguish the two classes. However, it is

[6]The term *bias* has several meanings in the field of machine learning. In case of the representational bias it can be understood as any factor that influences the selection of models or hypotheses [Gordon and Desjardins, 1995]. However, in case of the bias-variance dilemma the bias is the error between the prediction and the desired output (see section 2.2.2). In case of artificial neural networks the constant b is also sometimes referred to as bias (see section 2.1.2).

obvious that other vehicles will have different colors and that a classifier that relies only on the color will perform badly in practice.

This problem is especially important for learning tasks with only few training samples or noisy features. Classifiers might learn a model based on random noise, which somehow is sufficient to explain the training data but does not generalize. The resulting negative effects on the generalization performance are comparable to those from the bias-variance dilemma. However, in this case the problem does not originate from the selection of the classifier but the selection of the feature representation.

2.2.3 No-Free-Lunch Theorem

Section 2.1.2 presents some important and popular classifier concepts without directly denoting the "quality" of specific algorithms. The *no-free-lunch theorem* claims that there is no single best classifier that outperforms all other classifiers for all tasks. The theorem was formulated by [Wolpert and Macready, 1995] for search-based optimization problems and later for machine learning algorithms in [Wolpert, 1996]. Whenever optimization algorithms are compared, there is no universally best algorithm when all possible problems or datasets are considered. This means that very simple concepts like the naïve Bayes classifier or a single, linear perceptron may have a chance to outperform any other learning rule for a specific problem. The machine learning community has proposed a tremendous amount of different classifier concepts and variations of them which leads to the problem that it has become nearly impossible to compare every possible algorithm for each problem. Table 2.3 gives an overview of different classifiers and some properties regarding accuracy, speed and feature properties. Obviously no concept achieves the highest scores for all properties. In practice, a suitable classifier has to be selected for every task so that all demands are fulfilled.

2.2.4 Suboptimal Hyperparameters

Most machine learning algorithms are controlled by hyperparameters (see section 2.1.1) which have a great impact of the actual performance of the whole classifier. Consider the example of a multilayer perceptron (MLP, see section 2.1.2). Typically, an MLP has two hyperparameters, namely the number of hidden layers and the number of nodes on each hidden layer. These hyperparameters define the size and learning capacity of the network. The weights of the edges of the network are the model parameters that are adapted during the training phase. Other examples of hyperparameters are, e.g., the regularization hyperparameter c_{reg} of the C-SVM and the size γ_{Gauss} of a Gaussian kernel. The support vectors itself are the model parameters that are learned during training.

Criterion / Classifier	Rule Learners	Decision Trees	Neural Networks	Naïve Bayes	kNN	SVM
Accuracy in general	**	**	***	*	**	****
Speed of learning	**	***	*	****	****	*
Speed of classification	****	****	****	****	*	****
Tolerance to irrelevant attributes	**	***	*	**	**	****
Tolerance to redundant attributes	**	**	**	*	**	***
Tolerance to noise	*	**	**	***	*	**
Dealing with the danger of overfitting	**	**	*	***	***	**
Hyperparameter handling/sensitivity	***	***	*	****	***	*

Table 2.3: Properties of different classifier concepts according to [Kotsiantis, 2007]. The meaning of the scale is: $*$ worst and $****$ best performance. However, these properties are only a rough quality indication and certainly do not hold for all possible learning tasks.

The positive aspect of hyperparameters is that they allow an adaptation to the learning task, e.g., an easier task might require an MLP with less hidden layers than a complex task. On the other hand, every hyperparameter needs to be tuned and bad choices of hyperparameters degrade the performance. The problem is called hyperparameter optimization or model selection problem [Bergstra et al., 2011, Bishop, 2006]. This effect is explained with an MLP in the following. If the chosen network size is too small, the resulting classifier is too simple to learn a reasonable model. If the network size is too large, there are too many weights to train and the backpropagation learning algorithm is likely to get stuck in local optima. Additionally, large networks tend to overfit to the training data. These effects of hyperparameters are directly related to the bias-variance dilemma.

2.3 Discussion

Machine learning approaches are generally versatile and tempting as they theoretically provide a fast development of, e.g., classification systems. The main advantage is that no explicit modeling of the mapping between inputs and desired outputs is necessary. Instead, "just" some training data with ground truth labels has to be collected. Furthermore, powerful algorithms

like the SVM and random forests have been established that achieve a very good performance for a broad range of applications.

However, the list of machine learning challenges shows that the desired performance can be hard to reach in real-world applications. The diverse effects of the curse of dimensionality often cause a bad performance. Additionally, according to the no-free-lunch theorem, there is no generally best performing standard machine learning algorithm. Especially inexperienced users might use suboptimal combinations of features, algorithms or hyperparameters due to personal preferences, availability of code or lack of time. Furthermore, the selected training dataset might be too small or the chosen samples are not representative for the whole distribution.

The use of a machine learning system can even be dangerous, when critical or security-related applications are developed and the system performance is not properly estimated. Consider the application of autonomous vehicles for instance: A vision-based system that should automatically distinguish, e.g., cars from pedestrians needs to be designed and tested extremely carefully.

Chapter 3

Solutions to Improve Machine Learning

The previous chapter discusses well-known challenges of machine learning that degrade the overall performance in many real-world applications. Of course, the machine learning community has developed and established numerous suitable solutions for each aspect. This chapter discusses two different types of approaches to overcome the problems and is organized as follows.

The intuitive approach is the use of established or "standard" solutions, like cross-validation, feature selection and hyperparameter tuning that can be found in most textbooks about machine learning. Many of these methods rely on *heuristic optimization methods* as the underlying problems are complex. Therefore, section 3.1 provides a brief overview of heuristic optimization methods. The standard solutions for machine learning, such as feature selection, feature preprocessing and model selection, are presented in section 3.2. The second approach to improve machine learning is the field of representation learning which is discussed in section 3.3. Finally, in section 3.4 the challenges and solutions are connected to discuss the need for a holistic framework containing all aforementioned solutions.

3.1 Overview of Optimization Heuristics

Optimization methods are versatile mathematical tools that are used in all fields of research. Optimization plays a central role in the field of machine learning. The training of a classifier with its internal model parameters is also an optimization process. Some classifier concepts rely on practically well

solvable optimization problems. The SVM, e.g., uses a problem formulation that is convex[1] and can be solved with quadratic programming methods[2].

However, optimization problems like the selection of useful features (see section 3.2.2) or hyperparameters (see section 3.2.4) do not have "nice" properties, like smoothness[3] or convexity. Usually, it is very hard or even impossible to find the globally best solution, but it is often enough to find near-optimal ones. For these optimization problems, *heuristics* can find good solutions while only few assumptions are required about the problem itself. The problem is often considered as a *black box* and only an objective function is available whose output has to be either maximized or minimized. A general problem definition can be stated as

$$g_{Obj}(S_{\mathbf{h}}) = \mathbf{g} \in \mathbb{R}^{N_{Metrics}} \tag{3.1}$$

in which the objective function g_{Obj} is influenced by a variable set $S_{\mathbf{h}} = \{h_1, h_2, \ldots, h_{N_{Opt}}\}$ of N_{Opt} variables h_j. These variables can be of arbitrary type, e.g., continuous, real-valued variables. This set is also called solution vector. The output is a vector of metrics $\mathbf{g}$ while two cases are distinguished in general:

- In case of $N_{Metrics} = 1$, only one numeric value needs to be optimized which is called a *single-objective* problem. This is the "simplest" case as every minimization problem can be trivially formulated as maximization problem when $(-1) \cdot g_{Obj}(S_{\mathbf{h}})$ is considered.
- The case of $N_{Metrics} > 1$ is called *multi-objective optimization*.

In case of single-objective optimization, the best solution out of a set of candidates $S_{Candidates} = \{S_{\mathbf{h},1}, S_{\mathbf{h},2}, \ldots, S_{\mathbf{h},N_{Candidates}}\}$ can be selected simply by ordering their optimization metrics, e.g., $S_{\mathbf{h},1}$ is better than $S_{\mathbf{h},2}$, if

$$g_{Obj}(S_{\mathbf{h},1}) > g_{Obj}(S_{\mathbf{h},2})\,. \tag{3.2}$$

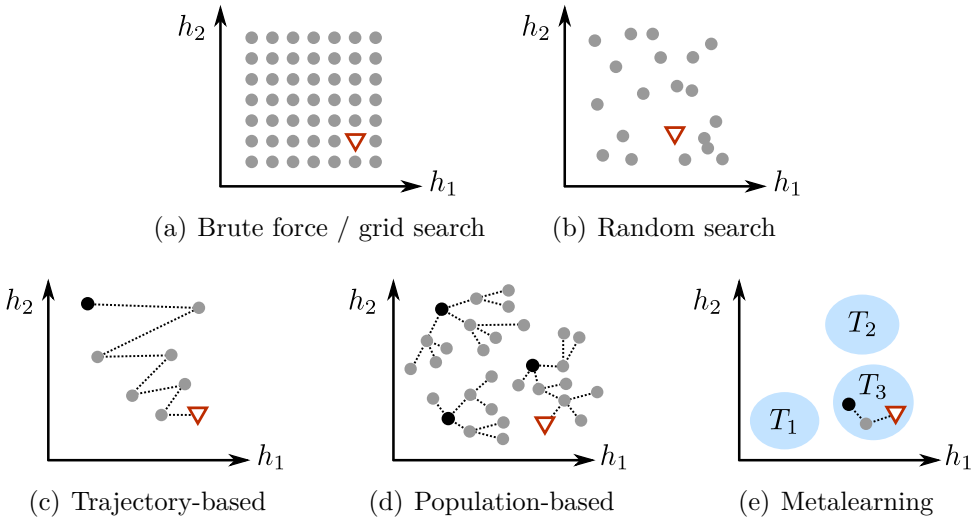
Multi-objective optimization is more challenging as metric vectors cannot simply be ordered by their values. In this case, a Pareto optimization[4] is required. However, this work focuses on single-objective optimization; for the multi-objective case the reader is referred to extensive textbooks like [Miettinen, 1999].

[1] Convex optimization problems only have global optima and thus optimization algorithms cannot get stuck in local optima.

[2] Quadratic programming methods are used to optimize a quadratic objective function of several variables in combination with a set of linear variable constraints.

[3] Smooth functions allow the use of the derivatives for methods like gradient descent.

[4] In Pareto optimization a set of the best solutions is aimed at so that a further improvement of a single metric leads to a deterioration of the other metrics.



(a) Brute force / grid search (b) Random search

(c) Trajectory-based (d) Population-based (e) Metalearning

Figure 3.1: Principles for different optimization heuristics for an exemplary problem with two variables. Initial solutions are denoted as black dots, intermediate solutions as gray dots. Final and best solutions are denoted with triangle symbols.

The number of optimization variables and the corresponding value domains $\mathfrak{H}_j$ of these variables define the *search space*

$$\mathfrak{H}^* = \mathfrak{H}_1 \times \mathfrak{H}_2 \times \ldots \times \mathfrak{H}_{N_{Opt}} \,. \tag{3.3}$$

The typical challenge is that the number of combinations $|\mathfrak{H}^*|$ explodes even for relatively small values of N_{Opt}.

3.1.1 Approaches of Optimization Heuristics

There are several approaches to tackle such optimization problems, which can be categorized as simple search-based algorithms and metaheuristics. Figure 3.1 depicts the principles of the different strategies. For more details and references the reader is kindly referred to the work of [Blum and Roli, 2003].

Simple Strategies

Simple and naïve strategies can be applied for small search spaces or in case the evaluation of the objective function is very fast. As solutions do not depend on each other, the numerous evaluations of the objective function can run in parallel on today's multi-core processor computers.

- *Brute force* or exhaustive *grid search* (see figure 3.1 (a)) is a very naïve approach in which all possible combinations of values are evaluated

through the objective function. Despite the fact that this usually becomes infeasible for $N_{Opt} > 2$, variables with real-valued numbers can only be sampled using a grid with a finite sampling density.

- *Random search* (see figure 3.1 (b)) generates a set of random solution vectors and evaluates them. This approach is considered as more efficient for a greater number of optimization variables. Furthermore, variables with real-valued numbers can be handled without the need of a coarse grid. However, a large number of evaluations is still necessary.

Metaheuristics

The simple approaches are usually not acceptable for real-world problems. Therefore, more intelligent *metaheuristics* have been developed. According to [Blum and Roli, 2003] these methods can be characterized as follows. Metaheuristics are guided search processes that efficiently explore the search space. Most of them are nature-inspired and contain the concepts of *diversification* and *intensification*. Diversification allows the efficient exploration of huge parts of the search space, which is usually achieved by random components. Intensification refers to the principle of local search to refine solutions.

Trajectory-based Metaheuristics

Trajectory-based heuristics start with an initial solution and try to follow a path – the trajectory – in the search space to refine the solutions. This principle is depicted in figure 3.1 (c). Established standard techniques are the following:

- *Simulated annealing* was introduced by [Kirkpatrick et al., 1983] and its basic idea is that the evaluation of inferior solutions is partly desired to be able to escape local optima. The probability of choosing worse solutions decreases over time – a principle that is inspired by thermal cooling processes.
- *Tabu search*, developed by [Glover, 1986], keeps a history of evaluated search space areas and forbids to "visit" them again. This allows the exploration of new areas in the search space and simultaneously the escape from local optima.

Model-based or Bayesian optimization [Brochu et al., 2010, Snoek et al., 2012] is a recently evolving approach to optimize difficult problems with many variables, especially with expensive – in terms of processing time – objective functions. *Sequential Model-Based Optimization* (SMBO) [Hutter et al., 2011] is the most promising variant of Bayesian optimization. The basic idea is to model the complete knowledge about the objective function and the uncertainty of unexplored areas of the search space as a probability distribution.

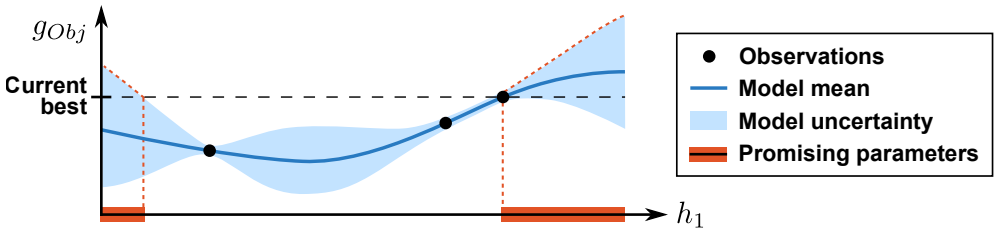


Figure 3.2: General principle of model-based or Bayesian optimization. A one-dimensional, numerical hyperparameter is considered and three observations in form of evaluations of the objective function g_{Obj} are made. A model function is trained with this data to predict mean and uncertainty of g_{Obj}. It is used to find promising areas of the search space that may improve the currently best solution. The figure is modified from [Brochu et al., 2010].

During the optimization process this knowledge is used to determine the next most promising variable configurations for the evaluation. Figure 3.2 shows this principle with a one-dimensional toy example. The goal is to save computation time on most likely suboptimal solution candidates, and so this approach is similar to tabu search in this respect. Early versions of this method use Gaussian distributions that are updated at each evaluation. The choice of Gaussian distributions limits this approach to numerical variables, however, extensions for categorical variables exist [Hutter et al., 2011]. These use, e.g., random forests to handle heterogeneous hyperparameters with numerical and categorical variables.

Model-based optimization is still a topic of ongoing research in the German Research Foundation (DFG) Priority Program "Autonomous Learning"[5]. Within the project "Auto-Tune: Structural optimization of Machine Learning frameworks for large datasets", the SMBO approach is currently used to optimize the hyperparameters of complex deep learning networks [Hutter et al., 2015].

Population-based Metaheuristics

Population-based algorithms use a set of candidate solutions in order to explore the search space. Usually, these algorithms coarsely start to cover the search space in order to find potential areas in which local refinement is the most promising (see figure 3.1 (d)). The evaluation of a set of candidate solutions is usually independent of each other. This leads to one of the central advantages of these approaches, namely the straightforward parallel implementation that benefits from today's multi-processor computers to speed-up the optimization.

[5]The Priority Program is referenced as DFG SPP 1527 *Autonomes Lernen*.

Evolutionary Optimization Algorithms

Evolutionary Algorithms are an important example of population-based approaches that imitate the evolution of species over time. The evolution theory of Charles Darwin [Darwin, 1859] describes the basic principles of this biological adaptation process in which only the fittest animals of a species survive within a changing environment. In the following, a brief description of the basic theory is provided.

The key is the concept of *genes* that contain and transport heredity information of each *individual*. Its total genetic information is contained in the *genotype* which is stored in the DNA of all living organisms. The location of this information within the genotype is called *locus*. However, not all genetic information has an effect on physical properties. The set of physical traits or characteristics of a specific individual is called *phenotype*, e.g., the color of the eyes.

Multiple individuals live in a *population* that is evolving over time. Occasionally, the genetic information of individuals is changed randomly by *mutation*. Each individual interacts with its environment and the degree of adaptation is called *fitness*. The *selection* principle states that only those individuals with the highest fitness have the chance to mate with other individuals of the current population and generate the offspring generation. Within the mating process, the genetic information of the involved individuals is fused by *recombination*. The recombination along with the mutation process create individuals with new properties that are potentially better – by means of fitness – than the parental generation. Individuals with worse properties will more likely die without generating offspring. Furthermore, individuals have a *limited age*, so that even the fittest individuals die after a certain number of generations. This mechanism increases the diversity within the population because non-dominant individuals with slightly lower fitness have the chance to evolve as well.

These biological principles have been adapted for the development of numerous variants of optimization algorithms, which can be found in many standard textbooks such as [Bäck, 1996, Eiben and Smith, 2003, Kramer, 2008]. First of all, the random character of this algorithm family has to be pointed out. The mostly random optimization process is guided by a certain subset and specific implementations of *evolutionary operators*. These operators are derived from the biological principles described above, namely mutation, selection and recombination. The large influence of randomness must not be seen as "stupid guessing" but as a chance to adequately cope with numerous local optima of the objective function. Furthermore, the different Evolutionary Algorithms use various genetic representations of the problem solution, e.g., sequences of Booleans, numerical variables or program code. In

the following, a brief overview of the most prominent variants of Evolutionary Algorithms is given, ordered by the year of publication.

- *Evolutionary Programming*, introduced by [Fogel et al., 1966], does not specify the structure of the genetic information. Only the mutation of individuals is involved, but no recombination as all individuals are considered as their own species that are not compatible with the others. The selection operator is implemented as a competition between groups of individuals in which scores are assigned. Finally, the individuals with the highest score are selected for the next generation.
- *Evolution Strategies* were introduced by [Rechenberg, 1973] and extended by [Schwefel, 1977]. Compared to other Evolutionary Algorithms, the representation and all operators are defined precisely. In the standard version the representation of a solution is a vector in $\mathbb{R}^{N_{Opt}}$, which is suitable to handle multiple numerical variables. The mutation operator is usually realized as an additive Gaussian-distributed noise vector and the recombination is done by averaging of the solution vectors. The current state-of-the-art for numerical parameter optimization are the so-called CMA-Evolution Strategies, the *Covariance Matrix Adaptation* [Hansen, 2006]. The recently developed approach of *Natural Evolution Strategies* [Wierstra et al., 2014, Glasmachers et al., 2010] combines Evolutionary Algorithms with gradient-based search approaches. Subsequent extensions also allow the handling of other variable types than real-valued variables, such as integers or categorical variables [Müller, 2012], which make Evolution Strategies more flexible.
- *Genetic Algorithms*, first introduced by [Holland, 1975], use a representation for the properties of the individuals which is very close to the DNA, namely sequences of Boolean variables – also called bitstrings. The variables of the optimization problem at hand need to be adequately coded as bitstrings, e.g., numerical hyperparameters require a suitable binary representation[6]. The fact that various representations behave differently regarding the mutation and recombination operators has to be considered carefully.
- *Genetic Programming*, introduced by [Koza, 1992], is the most flexible variant of Evolutionary Algorithms. The genome of each individual represents a computer interpretable program based, e.g., on context-free grammars, which can be treated as a tree structure. The mutation operator can affect the structure of the tree as well as the data that is stored in each node. The recombination operator can be implemented

[6] An established coding schema for floating point numbers is, e.g., defined by the IEEE 754 standard, which splits the number into a sign, a coefficient and an exponent. Integers can be coded in the two's complement representation.

in different ways, e.g., cutting and combining parts of the program. On the one hand, this approach can theoretically handle arbitrary problems as the representation allows the generation of any kind of program. On the other hand, it is a disadvantage that the generated programs are usually hard to understand for humans.

Other Population-based Metaheuristics

There are other established concepts of population-based optimization algorithms which have to be mentioned as well:

- *Particle swarm* algorithms, introduced by [James and Russell, 1995], are inspired by the natural principle of swarm intelligence and show similarities to Evolutionary Algorithms. Swarm intelligence describes a specific form of intelligence that is observed for some animal species: individuals are only able to perform relatively simple behavior patterns, however, a larger group of them can act surprisingly intelligent. Candidate solutions are represented with the positions of particles in a vector space. The particles also have a velocity that directs them into the direction of the best solutions.
- *Ant colony optimization* [Dorigo et al., 2006] is a variant of particle swarm approaches that uses a heuristic inspired by the behavior of ants. It has been observed that some species of ants deposit the scent pheromone along paths that lead to food sources. When other ants walk around, more likely they choose paths with a high concentration of pheromone. Algorithms that use this heuristic have been proved to be well suited to quickly find solutions for combinatorial problems.

Metalearning

Metalearning makes use of the knowledge of previous optimization or learning problems to improve the results on future tasks. This and very similar concepts are also known in literature under many different names: learning by analogy, transfer learning, learning to learn, life-long learning, knowledge transfer, inductive transfer, multi-task learning, knowledge consolidation and context-sensitive learning [Carbonell, 1983, Pan and Yang, 2010]. An all-encompassing overview about metalearning can be found in [Lemke et al., 2013]. A common principle is depicted in figure 3.1 (e) in which T_1, T_2 and T_3 denote areas of the search space with good variable values for previously solved tasks. In the example the current task is similar to T_3 so that a better initial solution is chosen for, e.g., a trajectory-based search heuristic. This can potentially lead to a speed-up of the optimization runtime and better results. The greatest challenge in metalearning is finding the best fitting pre-

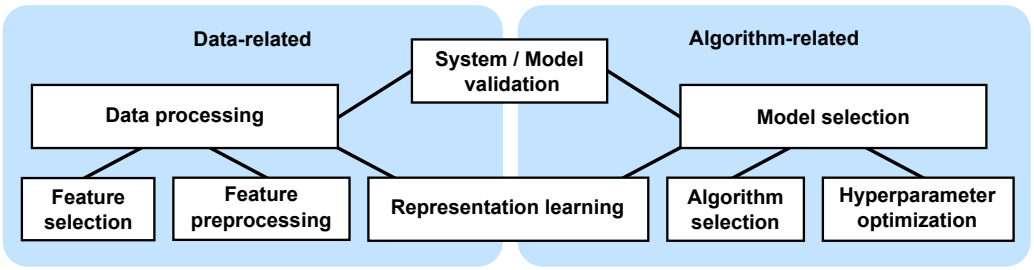


Figure 3.3: Taxonomy and connections of popular solutions for machine learning challenges.

vious tasks to the current task at hand. Section 3.2.5 describes this topic in greater detail.

3.2 Established Solutions for Improving Machine Learning

This section gives a rough overview of popular standard solutions which can be found in many textbooks such as [Bishop, 2006]. Figure 3.3 depicts the taxonomy of the different kinds of solutions which are popular to handle the typical challenges.

The solutions can generally be subdivided into *data-* and *algorithm-related* approaches. Data-related methods try to optimize the feature data itself before the classifier comes into play. Algorithm-related methods select and optimize the best fitting machine learning algorithm for a specific dataset with the aim to maximize the generalization performance. Some approaches cannot be clearly categorized as either data- or algorithm-related:

- System and model validation (see section 3.2.1) is always necessary to estimate the generalization performance of the selected machine learning system, consisting of the selected or derived features as well as the selected and tuned classifiers.
- Representation learning approaches (see section 3.3) calculate new features using a large variety of algorithms.

In the following subsections popular standard improvements for machine learning are presented, which are also included in many machine learning programming libraries and frameworks. These libraries are available for almost any programming language or platform and accelerate the use of machine learning algorithms tremendously.

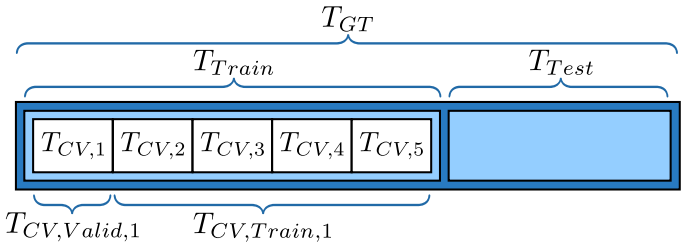


Figure 3.4: Dataset divisions for the cross-validation method using $N_{CV} = 5$. The compositions of the training and validation datasets ($T_{CV,Train,1}$ and $T_{CV,Valid,1}$) for the first cross-validation round are shown.

Publicly available program libraries are, e.g., the *WEKA library*[7] [Hall et al., 2009] for Java, *scikit-learn* [Pedregosa et al., 2011] and *mlpy* [Albanese et al., 2012] for Python and the *Shogun Machine Learning Toolbox* [Sonnenburg et al., 2010] for multiple programming languages. A popular example for a commercial software framework is the Matlab[8] *Statistics Toolbox* [The MathWorks, 2014]. The *Shark* C++ machine learning library [Igel et al., 2008] offers machine learning algorithms in combination with Evolutionary Algorithms for optimization.

3.2.1 System and Model Validation

System or model validation describes the estimation of the generalization or prediction performance of machine learning algorithms. It is a central step that needs to be made for the machine learning system as a whole, which covers the choice of specific algorithms, the set of hyperparameters and the selected feature set.

Machine learning systems need to be adapted for each learning task that is defined – in many cases solely – by the ground truth dataset T_{GT}. The dataset is subdivided into a training dataset T_{Train} and a test dataset T_{Test} (see section 2.1.3). In order to select and optimize an algorithm, only the training dataset T_{Train} may be used.

A generalization estimator $g_{est} : A \times T_{Train} \mapsto \mathbb{R}$ is a metric that assigns a quality measure to an algorithm A for a training dataset T_{Train}. This metric is used to compare two algorithms A_1 and A_2. If (without loss of generality) $g_{est}(A_1, T_{Train}) > g_{est}(A_2, T_{Train})$, algorithm A_1 will likely perform better on the learning task and – hopefully – any other previously unseen data from the same application.

[7] WEKA stands for "Waikato Environment for Knowledge Analysis".

[8] Matlab (MAtrix LABoratory) is a commercial computing environment developed by *The MathWorks, Inc.* with the focus on scientific computation. Its functionality can be extended by numerous so-called toolboxes.

A standard approach for a generalization estimator is *cross-validation* and the most commonly used variant is N_{CV}-fold cross-validation[9] [Bishop, 2006], which provides a fairly good estimation of the generalization at reasonable computational costs. Figure 3.4 shows the necessary divisions of the datasets for cross-validation. The training dataset is randomly subdivided into N_{CV} disjoint subsets with approximately[10] the same size so that

$$T_{Train} = \bigcup_{i=1}^{N_{CV}} T_{CV,i} \quad \text{with} \quad |T_{CV,i}| \approx |T_{CV,j}| \; \forall i, j \,. \tag{3.4}$$

Typical standard values for N_{CV} are 5 or 10 subsets. Subsequently, N_{CV} rounds of evaluations with $1 \leq i \leq N_{CV}$ are conducted in the following way. In the ith round, the subset $T_{CV,Valid,i} = T_{CV,i}$ is held out as a validation set while the training dataset is formed from the union of the rest $T_{CV,Train,i} = T_{Train} \setminus T_{CV,i}$. The classifier algorithm A is trained using $T_{CV,Train,i}$ and the predictions on $T_{CV,Valid,i}$ are evaluated using a performance metric like the overall accuracy (see section 2.1.3). Finally, the N_{CV} performance values q_i are needed to calculate the average performance

$$g_{est}(A, T_{Train}) = \frac{1}{N_{CV}} \sum_{i=1}^{N_{CV}} q_i \,, \tag{3.5}$$

which is used as an estimator for the generalization of the classifier. Finally, the most promising algorithm A with the highest value of g_{est} is picked.

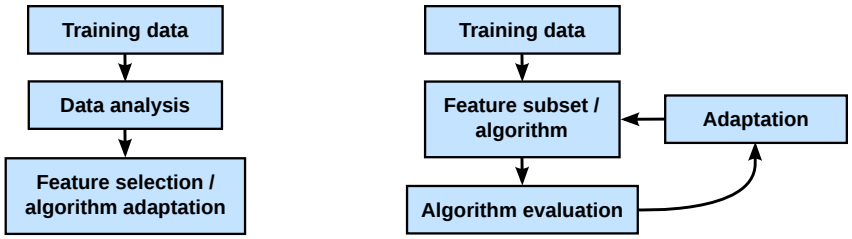
The advantage of cross-validation is that every evaluation is only performed on data that is not used for training. However, even the *selection* of an algorithm by means of cross-validation can lead to overfitting to the training dataset. The actual *generalization* performance of this model needs to be measured on the T_{Test} dataset afterwards which has never been used to select or train the algorithm model.

3.2.2 Feature Selection

Feature selection algorithms – also known as feature subset selection, variable selection or attribute selection – try to sort out irrelevant variables from feature data to obtain better performing machine learning algorithms. It is usually a very useful approach when the number of features is high and the classifier performance is degraded due to the curse of dimensionality. The review paper of [Guyon and Elisseeff, 2003] provides an overview and lists possible benefits especially for classifiers, i.e.

[9] Also well known as k-fold cross-validation.

[10] The sizes of the subsets can only be equal when the total number of training instances is a multiple of N_{CV}. However, it is usually sufficient when the the sizes of the subsets are only approximately equal.



(a) Filter approach (b) Wrapper approach

Figure 3.5: General principles of filter and wrapper methods for feature selection and (e.g. machine learning) algorithm adaptation.

- potentially improved generalization performance,
- less complex and faster classifiers,
- reduced measurement or feature calculation times and storage requirements and
- better understanding of the data and easier data visualization.

It has been reported that even classifiers that are well-known to handle high dimensionalities like the SVM can benefit from feature selection [Huang and Chang, 2007].

The feature selection problem is defined as follows. Given feature vectors with a dimensionality of D_{in}, the goal is to select a subset consisting of feature indices

$$S_{FeatSel} \in S_{all} = \mathcal{P}(\{1, 2, \ldots, D_{in}\}) \setminus \emptyset \tag{3.6}$$

that optimizes the classifier performance. The challenge is that the set of all possible subsets, defined by the powerset $\mathcal{P}(\cdot)$, grows exponentially with the number of dimensions as $|S_{all}| = 2^{D_{in}} - 1$. Consequently, it is usually impossible to consider all possible subsets even for relatively small values of D_{in}.

Algorithms for feature selection can generally be subdivided into two categories, namely *filter* and *wrapper* approaches. Note that these principles are also used for other problems such as algorithm adaptation, e.g., classifier optimization. Figure 3.5 provides an overview of the two approaches which are described in the following.

Filter Approach

A filter method (see figure 3.5 (a)) uses a single data analysis step to determine the feature subset or to adapt a, e.g., machine learning algorithm. The name originates from a filter or funnel that extracts only the essential information out of a large set of data. In case of feature selection, filters select the subset

$S_{FeatSel}$ solely based on properties of the feature data itself. This means that filter approaches are independent from the actual machine learning algorithm. A filter approach needs a metric that measures the relevance of each feature or alternatively ranks them by importance. The most common approaches and criteria are listed in [Guyon and Elisseeff, 2003], i.e.

- statistical correlation criteria such as the Pearson correlation coefficient, which detects linear dependencies between variables and labels,
- simple classifiers that only use a single feature for training to measure the predictive significance of each dimension and
- information theoretic metrics such as the data entropy to estimate the amount of information in each feature.

An alternative approach is the usage of classifiers that internally consider feature or variable importance ranking. The work of [Genuer et al., 2010] uses random forests (see section 2.1.2) in combination with a relatively simple method to obtain a robust and fast estimation of the variable importance. Each feature is tested individually by permutating[11] the feature values across all training samples. The expected resulting increase of the classification error is used as metric of importance. If a feature is important, a substitution of this feature dimension with random values will significantly increase the error. These metrics are averaged over all trees in the random forest and typically normalized to a range of $[0, 1] \subset \mathbb{R}$ to obtain a value $f_{FeatImp}(j) \in [0, 1] \subset \mathbb{R}$ for each feature $1 \leq j \leq D_{in}$. These importance metrics can be used to select the most important features easily, e.g., with a simple threshold operation

$$j \in S_{FeatSel} \Leftrightarrow f_{FeatImp}(j) > 0.5 \quad \text{for} \quad 1 \leq j \leq D_{in}\,. \tag{3.7}$$

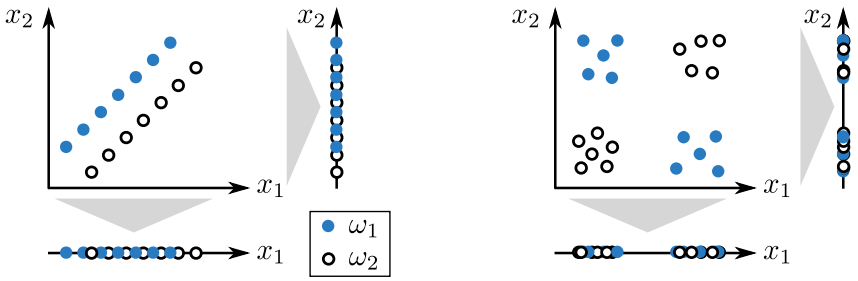
One major advantage of filters is that they are relatively fast. The computation complexity is usually $\mathcal{O}(D_{in})$ as these criteria are individually calculated for each feature. However, the disadvantages are obvious as combinations of features and the interplay with the final[12] classifier are not considered. Two relatively simple examples in which filters would fail to select reasonable variables can be found in figure 3.6. In the depicted scenarios single variables alone are rather useless, but the joint two-dimensional distributions are useful.

Wrapper Approach

Wrapper algorithms (see figure 3.5 (b)) use the desired, e.g., machine learning algorithm itself as a black box and evaluate multiple variants of it to

[11] The feature values in the particular dimension are randomly ordered so that the connection to the labels is lost.

[12] Some filters use the performance results of simple classifiers to estimate the importance of features. However, normally other, more complex classifiers are selected for the actual classification.



(a) Scenario 1 (b) Scenario 2

Figure 3.6: Usefulness of single variables compared to a joint distribution. In these two simple two-class scenarios a combination of two variables x_1 and x_2 can be useful (2D plot) while one variable alone is useless (1D axes). Overlapping class regions in the projected axes x_1 and x_2 prevent a proper distinction.

pick the best one. The name originates from the idea that this optimization approach "wraps around" any algorithm that should be optimized. In case of feature selection the wrapper algorithm evaluates the results of several feature subsets using, e.g., the average cross-validation accuracy. This leads to a feedback loop structure, which can also be found in control systems. It is an advantage that the interplay of mutual feature information and the classifier is considered. However, trying all possible combinations in a "brute force" manner is usually not feasible.

Several efficient, search-based heuristics have been proposed to find well-performing solutions within a reasonable time. Popular greedy strategies are *sequential selection methods*, which are subdivided into *forward selection* and *backward elimination*. Forward selection methods start with an empty subset and consecutively add items while backward elimination starts with the full feature set and sequentially removes features. These principles are used in the sequential forward and backward selection method which are abbreviated as SFS and SBS, respectively. More sophisticated algorithms are provided by the so-called *floating* extensions SFFS[13] and SFBS[14] [Pudil et al., 1994] that use backtracking when the current search direction becomes suboptimal. All these feature selection approaches are polynomial time algorithms.

Other, more sophisticated search heuristics can be applied as well. Most of the optimization metaheuristics described in section 3.1, such as Evolutionary Algorithms and particle swarms, are suitable. Many automatic optimization frameworks make use of these metaheuristics – see the examples given in section 3.2.5.

[13]SFFS stands for *Sequential Floating Forward Selection*.
[14]SFBS stands for *Sequential Floating Backward Selection*.

3.2.3 Feature Preprocessing

Raw feature or measurement data is often suboptimal for the direct usage in machine learning algorithms. Features may have completely different distributions with different value ranges. Consider the following example with two features. Feature x_1 lies in the range of $[0, 1] \subset \mathbb{R}$ with a small standard deviation while feature x_2 is spreading within the range of $[-10{,}000\,,\, +10{,}000] \subset \mathbb{R}$. If subtle changes in feature x_1 are highly relevant, classifiers that rely on canonical distances metrics will likely be disturbed by the large fluctuations in feature x_2. A practical example in the field of image processing are the Hu-Moments [Hu, 1962], a rotation and translation invariant image feature. The higher order moments tend to become very close to zero, e.g., smaller than 10^{-6}, which may cause numerical instabilities.

Several relatively simple feature preprocessing methods are established that eliminate these issues of the feature data, e.g., rescaling to a value range of $[0, 1] \subset \mathbb{R}$ or so-called pre-whitening to remove any inter-feature correlation in the data. Appendix B lists all relevant feature preprocessing methods with definitions and references. Feature preprocessing methods can be handled within a unifying feature preprocessing interface

$$A_{PreProc} = (f_{PreProc,estimate}, S_{Params,PreProc}, f_{PreProc})\,. \tag{3.8}$$

It consists of an estimation method

$$S_{Params,PreProc} = f_{PreProc,estimate}(T_{Train}) \tag{3.9}$$

that extracts model parameters $S_{Params,PreProc}$ from the training dataset T_{Train}, like minimum and maximum values for each dimension in case of rescaling. These model parameters are used to finally preprocess the feature vectors using

$$\mathbf{x}_{PreProc} = f_{PreProc}(S_{Params,PreProc}, \mathbf{x})\,. \tag{3.10}$$

3.2.4 Model Selection

The term "model selection" originates from the field of statistics in which probability distributions and their parameters need to be selected. However, in the field of machine learning it is used in a more "fuzzy" way: machine learning algorithms do not necessarily need to consider classical statistical probability distributions. Therefore, model selection can be generalized to the process of selecting a set of machine learning algorithms with the desired properties for a given task. This can be achieved in three ways:

First, only a single classifier concept, like the SVM, is considered and is tuned to the learning task – known as *hyperparameter optimization*. Secondly, a portfolio of multiple classifiers, like kNN, SVM and random forest, is considered and one of them is selected, which is called *algorithm selection problem*.

The third and most complex option is that a portfolio of classifiers is considered and additionally, the hyperparameters of the selected algorithm are tuned as well. This is called *algorithm configuration problem*. The following sections describe these three variants in detail.

Hyperparameter Optimization

The problem of hyperparameter optimization has long been recognized as an important field in machine learning but it is still an issue of active research. Section 2.2.4 describes the influence of hyperparameters and the necessity of their optimization. The most common hyperparameters can be categorized into three types:

- *Continuous, real-valued hyperparameters* can be defined as

$$H_{\mathbb{R}} = \left(h_{\mathbb{R}}, h_{\mathbb{R}}^{min}, h_{\mathbb{R}}^{max}\right) \tag{3.11}$$

  in which $h_{\mathbb{R}} \in \mathbb{R}$ is the actual hyperparameter value. The two values $h_{\mathbb{R}}^{min}$ and $h_{\mathbb{R}}^{max}$ define reasonable constraints for the value with $h_{\mathbb{R}}^{min} \leq h_{\mathbb{R}} \leq h_{\mathbb{R}}^{max}$ and $h_{\mathbb{R}}^{min}, h_{\mathbb{R}}^{max} \in \mathbb{R}$.

- *Integer hyperparameters* are similar but their values are integers

$$H_{\mathbb{Z}} = \left(h_{\mathbb{Z}}, h_{\mathbb{Z}}^{min}, h_{\mathbb{Z}}^{max}\right). \tag{3.12}$$

  The variable $h_{\mathbb{Z}} \in \mathbb{Z}$ is the hyperparameter value and lies between the constraints $h_{\mathbb{Z}}^{min} \leq h_{\mathbb{Z}} \leq h_{\mathbb{Z}}^{max}$ with $h_{\mathbb{Z}}^{min}, h_{\mathbb{Z}}^{max} \in \mathbb{Z}$.

- *Categorical hyperparameters* define an item out of a discrete set

$$H_{\mathbb{S}} = (h_{\mathbb{S}}, S_{Cat}) \tag{3.13}$$

  with $h_{\mathbb{S}} \in S_{Cat}$ and the order of the items in the set is not important. This type of hyperparameters is used for different options in algorithms, e.g., the choice of distance metrics in the kNN classifier.

Constraints for numerical hyperparameters are particularly useful to speed up the optimization process by focusing only on reasonable values.

The problem of hyperparameter optimization can be formulated as follows. An algorithm A has a set of $N_{Hyp}(A)$ hyperparameters

$$S_{\mathbb{H},A} = \{H_1, H_2, \ldots, H_{N_{Hyp}(A)}\} \quad \text{with} \quad H_j \in \{H_{\mathbb{R}}, H_{\mathbb{Z}}, H_{\mathbb{S}}\}, \tag{3.14}$$

which can be a combination of real-valued, integer and categorical types. The objective function

$$g_{Obj,Hyper}(A, S_{\mathbb{H},A}, T_{Train}) = \mathbf{g} \in \mathbb{R}^{N_{Metrics}} \tag{3.15}$$

estimates the generalization of the algorithm A considering the interplay of the training dataset T_{Train} and all hyperparameters. This is basically the same optimization problem definition which is described in section 3.1, but here the input parameters of the objective function are more specific.

The problem of hyperparameter optimization is usually difficult because of its combinatorial complexity and typically non-smooth objective functions (see section 5.2). In the following, different approaches to tackle the hyperparameter optimization problem are presented.

Simple Search Heuristics

Even today, in a considerable fraction of publications for practical applications of machine learning, classifiers are used without hyperparameter tuning. Manual hyperparameter tuning is also applied sometimes, however, it is inefficient even for a small number of hyperparameters. The simplest automation of this time-consuming manual work is the usage of *grid search*, which samples the hyperparameter values on a discrete grid. This method is very easy to implement with a few lines of code in every programming language. However, for a large number of hyperparameters this approach becomes infeasible because of the combinatorial explosion. Theoretical considerations and practical experiments in [Bergstra and Bengio, 2012] show that *random search* is more efficient than grid search for larger search spaces. For most cases of hyperparameter optimization these naïve solutions are suboptimal. Therefore, metaheuristics are used for a more efficient optimization.

Evolutionary Optimization

Evolutionary Algorithms have proved to be efficient for the optimization of multiple hyperparameters and can be extended to solve even more complex search problems. The authors of [Fröhlich et al., 2003], [Huang and Wang, 2006] and [Huang and Chang, 2007] successfully use Evolutionary Algorithms to optimize multiple hyperparameters of the SVM, namely the regularization parameter c_{reg} and kernel parameters γ_{Gauss}, along with the selection of features. The main differences lie in the coding schemes of hyperparameters and features. In [Åberg and Wessberg, 2007] this approach is extended to multiple classifiers, and tuned linear regression models are compared with non-linear artificial neural networks.

Other Metaheuristics

Besides Evolutionary Algorithms, other metaheuristics have been applied for multi-aspect machine learning optimization. The work of [Lin et al., 2008a] uses simulated annealing for simultaneous feature selection and hyperparameter optimization of the SVM. In [Lin et al., 2008b] the same authors also

successfully apply a particle swarm optimization approach to the same problem.

3.2.5 Algorithm Selection and Configuration

Usually, when not only a single classifier is considered but a portfolio of multiple algorithms, the problem is called *algorithm selection and configuration problem*. This issue especially arises in machine learning as there is no perfect learning algorithm due to the no-free-lunch theorem (see section 2.2.3). Algorithms are often selected manually based on personal knowledge, reputation or availability in program libraries. Furthermore, a lack of time often prevents the trial of alternatives. However, there is a large performance potential when several algorithms are compared. It is obvious that naïve approaches such as grid search are not efficient for such problems.

Metalearning

Pure search-based frameworks principally forget about the results of all previously solved problems and require the full amount of optimization runtime for each new problem. In contrast, metalearning uses knowledge of previous tasks for algorithm recommendation systems. A set of previously solved learning tasks and the knowledge of the best performing algorithms for each task are exploited to learn a *meta classifier*. This is usually done by deriving *meta features* from the training datasets such as statistical properties, e.g., the number of samples and dimensions, or so-called landmarking features[15]. Finally, the meta classifier learns the connection of the meta features to the best performing algorithms.

The work of [Reif et al., 2012] discusses several metalearning based algorithm recommendation systems that predict the performance of multiple classifiers with their hyperparameters. The focus of their work is the selection of meta features to achieve the best prediction of the actual classifier performance. This prediction can be used to select the best algorithm for a given dataset.

It is worth noting that the selection of meta features to predict the classifier performance should not be confused with feature selection of the dataset. There is no straightforward way to tackle the problem of feature selection of the dataset with the help of metalearning.

[15] Landmarking features use the performance, e.g., the accuracy, of simple and fast learning algorithms on the training datasets as a feature. These features are expected to be correlated to the problem complexity.

Bayesian Optimization

Bayesian optimization in form of sequential model-based optimization has recently become more popular for the optimization of complex search problems. The *Auto-WEKA* framework [Thornton et al., 2013] for the WEKA library solves the combined algorithm selection and hyperparameter optimization problem. They consider 39 different classifiers with all relevant hyperparameters in a hierarchical structure. Furthermore, a preprocessing step of different filter-based feature selection methods is involved. The main goal is to provide an interface which requires minimal user knowledge to choose a powerful machine learning system. As this idea is very similar to the goals of this work and the set of optimized aspects is large, the performance of the *Auto-WEKA* framework is compared to the proposed framework in the evaluations (see chapter 7).

Another very recent framework that uses Bayesian optimization in combination with metalearning is the *auto-sklearn* framework [Feurer et al., 2015b]. It automatically adapts multiple data preprocessing methods and classifiers from the *scikit-learn* machine learning framework [Pedregosa et al., 2011].

Evolutionary Algorithms

In [Ansótegui et al., 2009] Evolutionary Algorithms are used for general purpose algorithm configuration. The proposed algorithm uses several extensions of standard Evolutionary Algorithms to be able to handle complex optimization problems with heterogeneous hyperparameters and variable dependencies[16]. These variable dependencies are tackled with a tree structure that defines a special cross-over operator. Furthermore, the concept of genders is introduced to the Evolutionary Algorithm that allows multiple gender-specific fitness functions for multi-objective optimization. The work of [Ansótegui et al., 2009] proves that special Evolutionary Algorithms are potentially able to solve complex algorithm configuration problems. However, they have not been used in the machine learning context to optimize such complex algorithm combinations comparable to, e.g., the *Auto-WEKA* framework.

Local Search

The *ParamILS* framework [Hutter et al., 2009] is a general purpose solver for the development and application of algorithms with a large number of hyperparameters. The approach is based on local search from a set of initial solutions. The key of the *ParamILS* framework is that the time spent for evaluating each configuration is limited to optimize the runtime. The system

[16] An example for dependent variables is a hyperparameter of an algorithm which is only active when the particular algorithm is selected.

is successfully used for problems such as the propositional satisfiability problem (SAT), protein folding or university time-tabling. However, there are no reports on applications of the *ParamILS* framework in the field of machine learning.

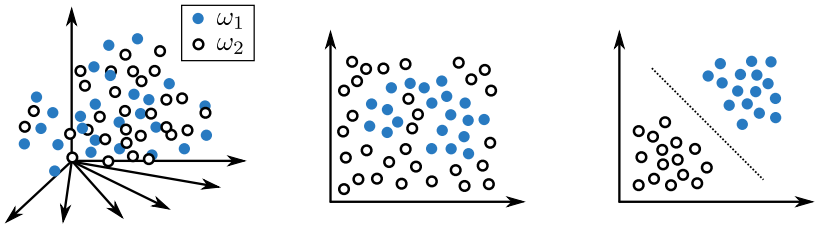
3.3 Representation Learning

The performance of machine learning algorithms is highly dependent on the suitability of the chosen features for a specific task. The features are the *representation* of the instances, which is the only information that is visible to the classifier algorithm when decisions are made. In many cases low-level data, such as sensor measurements or image data, needs to be used and the task is to correlate the raw data with desired class labels.

Figure 3.7 (a) shows typical raw data which is usually high-dimensional, noisy and the connection between the classes is complex and also hard to visualize. The typical approach is to manually choose or develop task-specific features or models that find better explanations for the data. This modeling is usually only possible with a lot of domain specific expertise. Depending on the complexity of the task, the quality of the raw data and the suitability of the chosen features, more useful representations such as depicted in figure 3.7 (b) or (c) are achievable. In case of classification, the aspect of *discriminative* features is essential for the performance of the system. The required complexity of a classifier model directly depends on the complexity of the data distribution and the boundaries between the classes. As complex, highly non-linear classifiers tend to overfit and become unstable, a linearly separable representation like in figure 3.7 (c) would be ideal.

Furthermore, the aspect of the *representational bias* (see section 2.2.2) has to be discussed in this regard. A suitable representation must not only explain the training data but also generalize to unseen samples. The risk of selecting an unsuitable representation is especially large for learning tasks with only few training samples and noisy features. This aspect is also connected to the feature selection problem (see section 3.2.2) as the removal of certain features might also remove information that would be necessary to classify unseen samples properly.

The research field of *representation learning* aims at automatically learning a better feature representation out of low-level data. An overview of the field is given by [Bengio et al., 2013] and one central aspect is to understand the general concepts of learning that explain abstract, underlying factors in-



(a) high-dimensional, high complexity (b) low-dimensional, medium complexity (c) low-dimensional, linearly separable

Figure 3.7: Different complexity levels of feature representations for a two-class classification problem.

side of data. These concepts are usually defined as *priors*[17] that should be universally valid across different learning domains. The authors of [Bengio et al., 2013] provide an overview of possible properties of such general purpose priors:

- *Smoothness:* A very basic assumption is that similar inputs lead to similar output values of a learned model, i.e. $x_1 \approx x_2 \Leftrightarrow f_{model}(x_1) \approx f_{model}(x_2)$. This prior is used in most machine learning algorithms but it is obviously too simple to be the only principle.
- *Hierarchy and abstraction:* This concept implies that data can be explained with different models in a hierarchy that leads to more abstract features in higher levels within the hierarchy. These concepts are used in *deep*[18] neural networks, which typically consist of hierarchical layers.
- *Semi-supervised learning:* This principle describes the usefulness of feature vectors without labels in combination with a usually smaller amount of labeled data. It is often the case that vast amounts of unlabeled data are available, which can improve the model of the feature distribution and thus help to explain the underlying concepts.
- *Transfer learning:* These concepts assume that explaining factors are shared across different datasets and problems. This is even more abstract than semi-supervised learning as the feature distributions may be completely different.
- *Clustering and manifolds:* These concepts make assumptions how the data is distributed in the feature space. Clustering exploits the fact that

[17]The term prior originates from the field of stochastic and refers to the choice of a specific probability distribution such as a Gaussian normal distribution. The term prior can also be interpreted as an assumption or model which is not limited to probability distributions.

[18]Deep learning uses large and deep network structures that learn more and more abstract concepts on each layer.

data with similar meaning tend to agglomerate in the same regions in the feature space. The concept of manifolds assumes that the data is embedded in regions with a lower dimensionality than the feature space (see section 3.3.2).

- *Sparsity and simplicity:* This concept makes assumptions regarding the properties of the derived representations themselves. Sparsity assumes that only a few factors of the representation are different from zero simultaneously. Simplicity implies that the dependencies between factors are simple, i.e. linear.

So far these principles of representation learning are rather theoretical and abstract and it is not yet obvious how to make use of them for real-world applications. What is needed are practically and constructive approaches that lead to a higher performance for machine learning algorithms.

The field of automatic *feature construction* provides this practical aspect and is therefore particularly interesting. A perfect feature construction algorithm would be able to automatically transform low-level features like shown in figure 3.7 (a) to a representation whose complexity is somewhere between figure 3.7 (b) and (c). Such a transformation could be used as a preprocessing step and would basically move parts of the "intelligence" away from the classifier. *Manifold learning*, *dimensionality reduction* methods and *feature transforms* are promising realizations of automatic feature construction, which are discussed in the following. First, section 3.3.1 presents recent findings about biological evidences for manifold-like representations and dimensionality reduction in the human brain, which serves as a motivation for representation learning. Section 3.3.2 presents an overview of practically useful algorithmic approaches to feature construction.

3.3.1 Manifold-like Representations in the Brain

Many machine learning approaches are inspired by structures that appear in the brain, e.g., artificial neural networks that make use of simple neuronal cell models. Simple, but highly interconnected neuron models can perform complex learning tasks (see section 2.1.2). However, the exact functionality of the brain is not fully understood yet.

One particular example is the human vision system that has extraordinary pattern recognition abilities. We can effortlessly see, recognize and distinguish tens of thousands objects like letters, numbers, cars, or faces within the fraction of a second. We can also learn new categories of classes of objects quickly just by seeing very few examples. Even more remarkable is the fact that the visual recognition is widely invariant to the huge variation of appearances which are possible for each object. These variations can be categorized as positioning, scaling, orientation, background context or clutter,

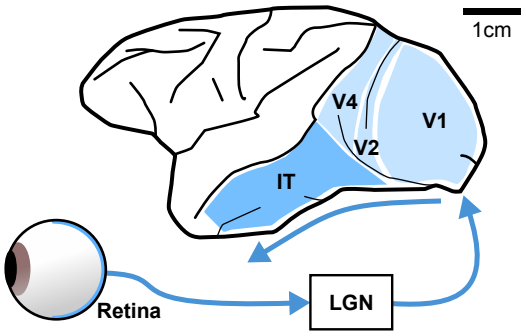


Figure 3.8: Flow of visual information in the macaque monkey brain from the retina to the visual cortex (V1, V2, V4) to the inferior temporal cortex (IT) according to [DiCarlo et al., 2012].

partial occlusion as well as different shading or colors of the perceived objects [DiCarlo et al., 2012]. This invariance is highly related to the desired high generalization capabilities of machine learning systems.

The ability of invariant object recognition is defined as the accurately identification of an object from all other possible objects. The first aspect is the detection of relevant objects. In order to achieve this, the vision system performs visual search, a fast preattentive mechanism to filter out distracting or uninteresting items [Wolfe, 1994]. The eyes quickly move around in saccades[19] to scan the scene for relevant objects or structures. There is a scientific consensus that visual search is based on simple features such as color, edges, orientation or movements and is realized in the early vision areas.

Study of the Monkey Brain

The species of humans, the *homo sapiens*, belongs to the primate order and our brain's general structure partly resembles those of nonhuman primates such as monkeys. The brain of the rhesus macaque monkey is the currently best model to understand the human brain and has widely been studied [DiCarlo and Cox, 2007]. Like humans, the perception of monkeys highly relies on vision and a huge part of their brain is dedicated to process visual stimuli and object recognition. Figure 3.8 shows the schematic structure of the visual system and their locations in the brain of the macaque monkey. The photons of the light reflected by an object are projected to the retina and stimulate around 100 million retinal photoreceptors. These signals are processed by around one million retina ganglion cells and these stimuli are

[19] Saccades are the movements of the eyes between glancing at fixed points.

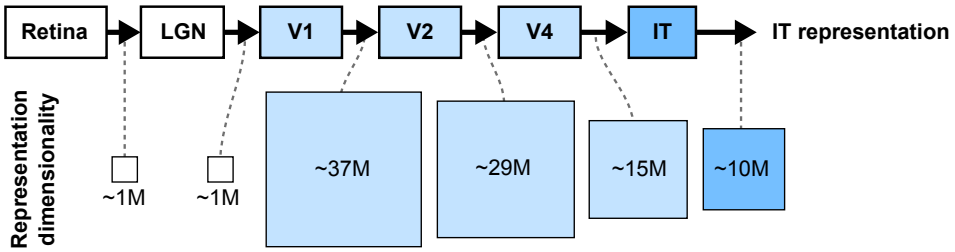


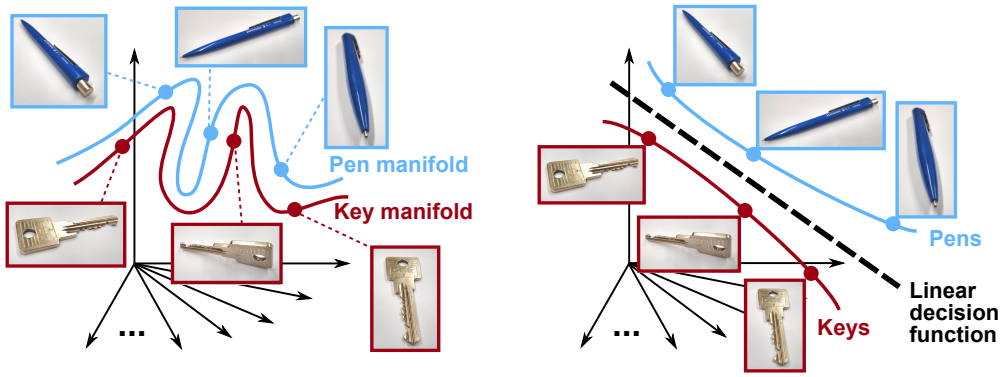
Figure 3.9: Feedforward processing of visual information in the visual system of the macaque monkey brain during object recognition according to [DiCarlo et al., 2012]. The approximate dimensionality (M = million) of the neural representations after each stage is visualized with the size of the squares underneath.

sent to the lateral geniculate nucleus (LGN) – located in the thalamus[20] – in form of spiking patterns. After being sent to the LGN the signals are transmitted to the visual cortex, more precisely to the subareas V1 , V2 and V4. Finally, the inferior temporal cortex (IT) processes the signals from the visual cortex. The IT area has been identified to be crucial for the object recognition task in the brain as this area shows a high activity when objects are visually perceived.

The general understanding of the object recognition problem is the following. Whenever an image of an object is perceived by the retina, the visual system has to transform this high-dimensional stimulus in form of millions of "pixels" into read-out[21] or classifier neurons that are activated when the object is recognized [DiCarlo and Cox, 2007]. Figure 3.9 shows the feedforward information processing from the retina to the IT representation. The representation dimensionality of each stage is estimated based on neuronal densities [Collins et al., 2010] and gives a rough insight into the processing complexity. The dimensionality of the early retina and LGN signals is increased by factor 37 in the V1 representation. One explanation is that a feature generation process takes place that generates features with a certain degree of redundant information. Looking further, the dimensionality of the representation is decreased consecutively within the visual cortex to around 15 million after V4 and, finally, the IT area has only 10 million neurons.

[20]The thalamus is situated in the center of the brain. It is responsible for multiple functions and can be understood as a "hub" that distributes information between different brain areas.

[21]This concept is related to the so-called "grandmother cell", which is a hypothetical neuron in the brain that is activated when a specific person is perceived [Gross, 2002].



(a) Representations in early areas

(b) Representations in IT space

Figure 3.10: Simplified visualization of two object manifolds (keys and pens) in different areas in the visual system. A manifold is formed by the neural response patterns of an object undergoing identity preserving transformations. Manifolds of early visual areas (a) – retinal ganglion cells, LGN, V1 – are highly folded and tangled, while the response patterns in the IT space (b) can be separated using a linear decision boundary.

Manifold Untangling Theory

A relatively new approach to understand the brain's functionality is the hypothesis of "untangling object manifolds" as described in [DiCarlo and Cox, 2007] and [DiCarlo et al., 2012]. The activity of a single neuron is determined by its spiking rate, which can be interpreted as a real-valued number. Given a population of $N_{Neurons}$ neurons, e.g., all neurons in the V1 area, the signal or response of the population can be seen as an $N_{Neurons}$-dimensional vector. In the earliest stage of visual signal processing an object is perceived as an image with a pixel representation. A high-resolution pixel-like representation is extremely high-dimensional and consists of low-level data. When viewing an object in different poses or under different conditions – the so-called identity preserving transformations – the pixel pattern is likely never the same on the retina. When all the activation patterns of an object undergoing all possible transformations are analyzed in the response vector space, they lie on a relatively low-dimensional manifold curve.

Figure 3.10 (a) shows manifold curves for two different objects, keys and pens, as simplified one-dimensional lines. In reality the dimensionalities of these vector spaces lie in the range of millions. In early areas of the visual system manifolds of different objects are highly curved and tangled but do not intersect each other. Interestingly, evidence has been found that in the IT response vector space the manifolds are "untangled" and flattened as figure 3.10 (b) shows. In [Hung et al., 2005] the spiking activity of the IT area of the macaque monkey has been measured using multi-unit activity (MUA)

recording[22] while objects have been shown to the animals. These signals could be used to train a linear classifier to categorize objects with an accuracy of 94% and non-linear classifiers did not perform any better. However, simple linear classifier models fail when earlier stages of the visual system are used since the problem is highly non-linear in these representations.

To summarize, invariant object recognition is a complex problem and the exact processing in the brain is not yet understood. The brain uses several stages of neuronal signal processing to perform object recognition. According to the object manifold hypothesis presented in [DiCarlo and Cox, 2007] the brain uses transforms to change the representation of highly tangled manifolds to simpler and flattened manifolds. It is not yet evident how the brain achieves these transforms and how artificial models could look like. However, the key lies in changing the representation from a complex, highly non-linear problem to a simpler, linear one. Furthermore, the deep and feedforward structure of the brain after the V1 area shows a consecutive dimensionality reduction even if the dimensionalities are still extremely high.

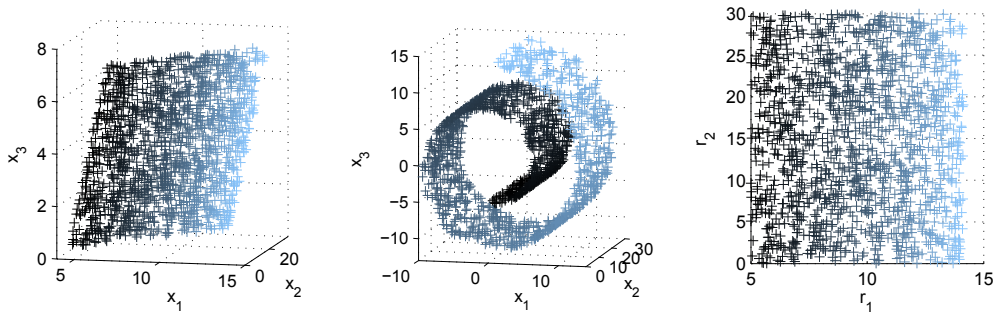
3.3.2 Manifold Learning and Dimensionality Reduction

Manifold learning describes a family of linear and non-linear dimensionality reduction techniques. The idea is to exploit the topological properties of data distributions to find a lower-dimensional and simpler representation of the high-dimensional data. Manifolds can basically be interpreted as geometrical distributions with a lower dimensionality than the vector space they are embedded in. Consider the following two examples of manifolds in a three-dimensional space. Figure 3.11 (a) shows points that are distributed on a two-dimensional linear plane that could be mapped into a two-dimensional coordinate map depicted in figure 3.11 (c) without losing information about the distribution. Figure 3.11 (b) shows an example of a non-linear manifold known as "Swiss roll"[23] dataset. Humans can quickly see that this shape could be unrolled to a two-dimensional plane as in figure 3.11 (c).

The task for a manifold learner would be to transform high-dimensional data shown in figure 3.11 (a) and (b) to a lower-dimensional representation depicted in figure 3.11 (c). Manifold learning has basically two applications with different goals:

[22] Multi-unit activity (MUA) recording is a measurement method to monitor electrical activity in discrete cell areas using electrodes. The brain activity of a specific region can be measured by implanting these sensors onto the brain tissue that should be analyzed [Windhorst and Johansson, 1999].

[23] The Swiss roll dataset is a popular artificial benchmark and is generated using a mapping of $[r_1, r_2] \mapsto [x_1, x_2, x_3]$ with $r_1 \in [3\pi/2, 9\pi/2] \subset \mathbb{R}$, $r_2 \in [0, 30] \subset \mathbb{R}$, $x_1 = r_1 \cos r_1$, $x_2 = r_1 \sin r_1$ and $x_3 = r_2$.



(a) Linear manifold on a plane in 3D

(b) Non-linear "Swiss roll" manifold in 3D

(c) Corresponding 2D coordinate map of (a) and (b)

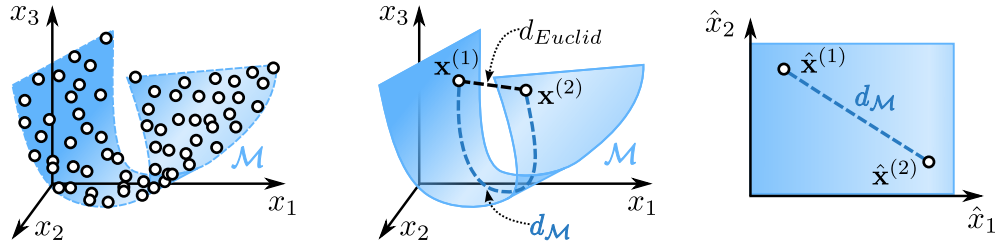
Figure 3.11: Examples of points on linear and non-linear manifolds in 3D with the corresponding underlying 2D coordinate map. Note that the shading of the sampled points indicates the mapping between the manifolds (a) and (b) to the 2D coordinate map (c).

1. *Visualization* helps to understand the distribution and explanatory factors of data in case of manual analysis. However, visualization is naturally limited to low-dimensional data – typically less than four dimensions[24] are used. In this case, manifold learning can be used to reduce the dimensionality to display the most important factors in a few dimensions.
2. *Feature construction* describes the automatic generation of a more suitable feature set, which is strongly related to representation learning. A representation based on the structure of a lower-dimensional manifold might not only be more understandable for humans but also potentially more useful for machine learning algorithms. Furthermore, the negative effects of the curse of dimensionality are dampened, if the dimensionality is reduced considerably.

Manifold Learning Definitions

Overviews of the field of manifold learning can be found in [Ma and Fu, 2011] and [Van der Maaten et al., 2009]. However, the exact definitions of manifold learning are not consistent in literature. First, the term *manifold* itself needs to be defined in a useful way. The term manifold was introduced by Georg Friedrich Bernhard Riemann (1826 – 1866) and originates from the German word *Mannigfaltigkeit*. Mathematically strict definitions can be found in [Ma and Fu, 2011] that basically consider properties of D_{low}-dimensional subspaces that are embedded into a D_{high}-dimensional space with $D_{high} \geq D_{low}$. For the

[24] Four or more dimensions can be visualized with a 3D plot and the usage of colors or different marker types.



(a) Learning a manifold $\mathcal{M}$ from data points

(b) Euclidean and geodesic distance

(c) Low-dimensional space based on the manifold $\mathcal{M}$

Figure 3.12: Visualization of the main steps of idealized manifold learning. The transformed coordinates are denoted as $\hat{\mathbf{x}}$.

sake of simplicity, the following definitions are restricted for standard real-valued coordinate spaces ($\mathbb{R}^D$) as they practically appear in machine learning.

- *Topological manifolds* are generalizations of curved surfaces in three-dimensional spaces. These spaces can be considered as sets that have at least local[25] properties of D_{low}-dimensional Euclidean spaces. Topological manifolds in the same vector space are considered as *disconnected* when the intersection of them is empty.
- A *Riemannian manifold* $\mathcal{M}$ is a topological manifold that is *smooth* in terms of being differentiable to any order. This property allows the definition of useful metrics on the manifold itself.
- A *metric* on a Riemannian manifold $\mathcal{M}$ is denoted as $d_{\mathcal{M}}(\mathbf{x}^{(1)}, \mathbf{x}^{(2)})$ and is defined as the length of a one-dimensional and differentiable curve entirely on the manifold that connects any two points $\mathbf{x}^{(1)}, \mathbf{x}^{(2)} \in \mathcal{M}$.
- *Geodesics* describe the shortest curve between two points on a manifold and the corresponding length of the curve is called *geodesic distance*. Geodesic distances are of special interest as they are potentially more meaningful for machine learning applications than, e.g., standard Euclidean distances in D_{high} dimensions. Figure 3.12 (b) depicts the difference of the geodesic and the Euclidean distance.

The model parameters of an abstract manifold $\mathcal{M}$ have to be estimated by analyzing the distribution of the feature vectors in the training dataset. This process is visualized in figure 3.12 (a) in which the samples are used to estimate a geometrical model of $\mathcal{M}$. Once this model is available in figure 3.12 (b), the geodesic distance $d_{\mathcal{M}}$ can be estimated. This is used by many manifold learning algorithms to project the points into a lower-dimensional space depicted in figure 3.12 (c). The geodesic distance is the shortest Euclidean distance within the new coordinate space.

[25]Some parts of a set of vectors have other properties than the rest of the vectors.

Generalizations for Dimensionality Reduction

In practice there are basically two issues with the strict and rather theoretic definition of manifold learning. First of all, real-world data is usually noisy and can have arbitrary complex distributions, which make the strict Riemannian assumptions unsuitable. Secondly, there are other dimensionality reduction techniques that directly transform points into a lower-dimensional space without calculating manifolds or geodesic distances explicitly. In a review of dimensionality reduction techniques [Van der Maaten et al., 2009] the author weakens the necessary assumptions for practical dimensionality reduction algorithms such that the manifolds may be non-Riemannian and thus contain discontinuities or several disconnected submanifolds. Consequently, a more general definition is needed that can be used as a unifying interface for any manifold learning, feature transform or dimensionality reduction algorithm. A *feature transform interface* is defined as a tuple of

$$A_{Trans} = (f_{LearnTrans}, S_{\mathbb{H},A_{Trans}}, S_{TransParams}, f_{Transform}) \tag{3.16}$$

consisting of the following components:

- A function $f_{LearnTrans}$ that is responsible for learning model parameters of the manifold or feature distribution based on the training dataset and the set of hyperparameters $S_{\mathbb{H},A_{Trans}}$ that controls the learning algorithm.
- The set $S_{TransParams}$ contains arbitrary model parameters that are necessary to store the manifold or transformation derived from the training data.
- The function $f_{Transform}$ is needed to transform sample vectors from the input feature space into the typically lower-dimensional target feature space.

The functionality of this interface is defined by two equations. First, the learning process is described as

$$S_{TransParams} = f_{LearnTrans}(T_{Train}, D_{low}, S_{\mathbb{H},A_{Trans}}) \tag{3.17}$$

and uses the training dataset T_{Train}, the *target dimensionality* D_{low} as well as the hyperparameter values $S_{\mathbb{H},A_{Trans}}$. The target dimensionality D_{low} is an important hyperparameter for almost all feature transforms and is therefore listed separately. In the optimal case D_{low} should be equal to the intrinsic dimensionality of the manifold. However, this intrinsic dimensionality is usually not known for real-world datasets. Therefore, the target dimensionality has to be estimated in a range of $1 \leq D_{low} \leq D_{high}$.

The learned set of model parameters is used to transform a vector $\mathbf{x} \in \mathbb{R}^{D_{high}}$ into the new space using

$$\hat{\mathbf{x}} = f_{Transform}(S_{TransParams}, \mathbf{x}) \in \mathbb{R}^{D_{low}}. \tag{3.18}$$

The feature transform function $f_{Transform}$ is also known as *out-of-sample extension* or *embedding*. This is an important aspect of all dimensionality reduction techniques that should be used for machine learning because previously unseen samples must be considered as well. Otherwise, these feature transforms would lead to a system that could only recognize the training dataset. The next section discusses several algorithms as well as the influence of properties of their out-of-sample extension.

Feature Transform Algorithms

Numerous different approaches have been proposed within the past decades that fit to the proposed feature transform interface. It would exceed the scope of this work to describe even the most relevant algorithms in detail so that the reviews of [Ma and Fu, 2011], [Van der Maaten et al., 2009] and [Lin and Zha, 2008] are referenced at this point. There is also no consensus about a straightforward taxonomy of feature transform methods as they have been developed for completely different applications such as visualization, statistics or machine learning. Instead, the following central properties of feature transform algorithms can be pointed out:

- *Type of optimization problem:* One possible categorization is proposed by [Van der Maaten et al., 2009], which distinguishes between the type of optimization problem that needs to be solved in the learning functions $f_{LearnTrans}$. Figure 3.13 depicts the proposed hierarchy that is split up into *convex* and *non-convex* optimization problems.

  Feature transforms with convex optimization problems often consider distance matrices that contain pairwise distance metrics between all feature vectors of the training dataset

$$\mathbf{D} = \begin{bmatrix} d(\mathbf{x}^{(1)}, \mathbf{x}^{(1)}) & d(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}) & \dots & d(\mathbf{x}^{(1)}, \mathbf{x}^{(N_T)}) \\ d(\mathbf{x}^{(2)}, \mathbf{x}^{(1)}) & d(\mathbf{x}^{(2)}, \mathbf{x}^{(2)}) & \dots & d(\mathbf{x}^{(2)}, \mathbf{x}^{(N_T)}) \\ \vdots & \vdots & \ddots & \vdots \\ d(\mathbf{x}^{(N_T)}, \mathbf{x}^{(1)}) & d(\mathbf{x}^{(N_T)}, \mathbf{x}^{(2)}) & \dots & d(\mathbf{x}^{(N_T)}, \mathbf{x}^{(N_T)}) \end{bmatrix} \in \mathbb{R}^{N_T \times N_T}. \tag{3.19}$$

  In many algorithms an eigendecomposition[26] of this matrix – also known as *spectral* analysis – is performed. Examples of these transforms are Local Linear Embedding (LLE) [Roweis and Saul, 2000] and Isomap

[26] An eigendecomposition of a matrix calculates its eigenvectors and eigenvalues.

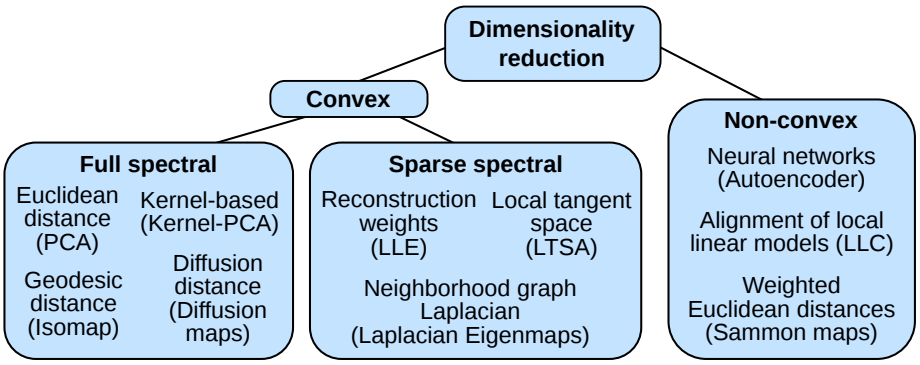


Figure 3.13: Taxonomy of dimensionality reduction techniques according to [Van der Maaten et al., 2009], which is based on the type of the optimization problem. One important example algorithm is listed in brackets under each category (see appendix C for the abbreviations).

[Tenenbaum et al., 2000]. This eigendecomposition can be either *full spectral* (analyze the global structure of the manifolds) or *sparse spectral* (focus on local structures).

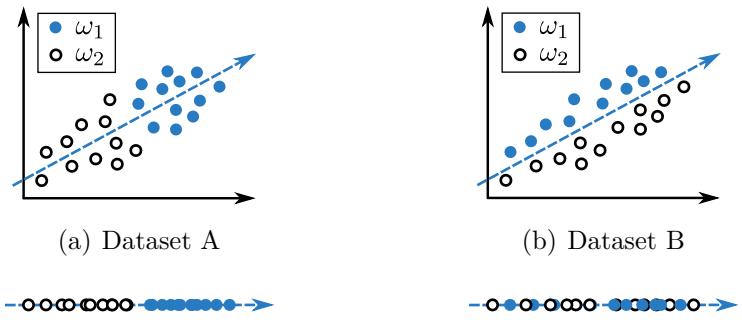
- *Linearity:* Linear dimensionality reduction methods basically learn a linear projection matrix that transforms high-dimensional vectors to the low-dimensional space using

$$\hat{\mathbf{x}} = \mathbf{x} \cdot \mathbf{W}_{lin} \quad \text{and} \quad \mathbf{W}_{lin} \in \mathbb{R}^{D_{high} \times D_{low}}. \tag{3.20}$$

  Important examples of linear projections are, e.g., the Principal Component Analysis (PCA) [Pearson, 1901] or the Linear Discriminant Analysis (LDA) [Fisher, 1936]. In contrast, non-linear methods perform transformations that cannot be expressed as such a matrix.

  Linear transforms have the general advantage that the projection itself can be processed very fast. However, the learning of the projection matrix $\mathbf{W}_{lin}$ can be of arbitrary complexity. Furthermore, simple and popular methods such as PCA tend to be very robust to noise within the training data. On the other hand, linear models might be too simple to describe the distribution of real-world datasets appropriately.

- *Supervision:* Most algorithms are *unsupervised*, so they build a model of the feature vector distribution that is independent from any labels. In the application of supervised classification, the labels $y^{(i)}$ provided in the training dataset are simply ignored. Unsupervised methods can make use of vast amounts of unlabeled data which is a central aspect of deep learning approaches. On the other hand, the label distribution can have a significant impact on the usefulness of the feature transform as depicted in figure 3.14.



(a) Dataset A (b) Dataset B

(c) PCA projection of A (d) PCA projection of B

Figure 3.14: Usefulness of an unsupervised PCA projection depending on the class label distribution. Note that the class labels are swapped from right-left to top-bottom between the datasets A and B. The projection in figure (c) simplifies the classification problem while the one in figure (d) is almost unusable as the class regions are overlapping.

- *Out-of-sample extension:* An out-of-sample extension of feature transforms is absolutely needed to process unseen samples for classification tasks. Unfortunately, not all feature transforms provide a straightforward way to embed new samples. Three categories of out-of-sample extensions can be distinguished:

  (a) *Direct out-of-sample extension:* This group contains feature transforms that provide a straightforward and usually fast way to embed new feature vectors into the low-dimensional space. These nice properties are provided by *linear* and so-called *parametric* methods. Linear methods are the simplest as they just require a matrix multiplication, like PCA. Parametric methods are usually more complex but have a straightforward transformation function $f_{Transform}$ based on learned model parameters. Examples are dimension-reducing feedforward neural networks such as Autoencoders [Hinton and Salakhutdinov, 2006].

  (b) *Approximation out-of-sample extension:* Some feature transforms do not originally provide a practically useful way to embed previously unseen feature vectors into the new space. Instead, approximations have been proposed for some methods that allow an out-of-sample extension. Important examples are methods that rely on a spectral eigendecomposition of a distance matrix between the training samples, e.g., LLE and Isomap. The authors of [Bengio et al., 2004] provide an approximation based on the Nyström method that basically assumes the convergence of the eigenvectors for a sufficiently large amount of training data.

(c) *No out-of-sample extension:* There is a variety of feature transforms which has been developed for visualization purposes and can be considered as "dead end" for machine learning due to the lack of a parametric extension function $f_{Transform}$. An example is Sammon's Mapping [Sammon, 1969]. In order to exploit a potentially useful representation for classification, a rather naïve non-parametric out-of-sample extension based on nearest neighbors can be used, which is provided by [Van der Maaten et al., 2009] and [Van der Maaten, 2014]. It requires the storage of the complete set of untransformed vectors and their corresponding mapped vectors $\{(\mathbf{x}^{(i)}, \hat{\mathbf{x}}^{(i)})\}$. The index i^* has to be found so that $\mathbf{x}^{(i^*)}$ of the untransformed vectors has the minimum Euclidean distance to the new sample $\mathbf{x}$. Then a linear transformation matrix

$$\mathbf{W}_{extension} = (\mathbf{x}^{(i^*)} - \text{meanVector}(\mathbf{x}^{(i^*)}))^{+} \cdot (\hat{\mathbf{x}}^{(i^*)} - \text{meanVector}(\hat{\mathbf{x}}^{(i^*)})) \quad (3.21)$$

is derived in which $\mathbf{x}^{+}$ is the Moore-Penrose pseudoinverse[27] and

$$\text{meanVector}(\mathbf{x}) = [\underbrace{\text{mean}(\mathbf{x}), \text{mean}(\mathbf{x}), \ldots, \text{mean}(\mathbf{x})}_{D \text{ times}}] \in \mathbb{R}^D \quad (3.22)$$

is a vector containing the mean of all components of a vector $\mathbf{x} \in \mathbb{R}^D$. This transformation is used to describe the difference vector of the untransformed feature space in the lower-dimensional target space. The embedding of $\mathbf{x}$ into the manifold space is realized by

$$\hat{\mathbf{x}} = (\text{meanVector}(\hat{\mathbf{x}}^{(i^*)}) + (\mathbf{x} - \text{meanVector}(\mathbf{x}^{(i^*)})) \cdot \mathbf{W}_{extension})^{\top}. \quad (3.23)$$

This method works for any feature transform, however, the very naïve approach will certainly cause estimation errors because it relies on nearest neighbors based on the Euclidean distance. Additionally, it is rather slow due to the neighbor search and the need to store the whole training dataset and its transformed vectors.

To achieve a unifying definition for the out-of-sample extension, the transform function $f_{Transform}$ is defined as either the direction extension, the approximation method or the naïve nearest neighbor estimation, depending on the availability.

Appendix C lists numerous feature transformations, which are used within this work, as well as references and their central properties.

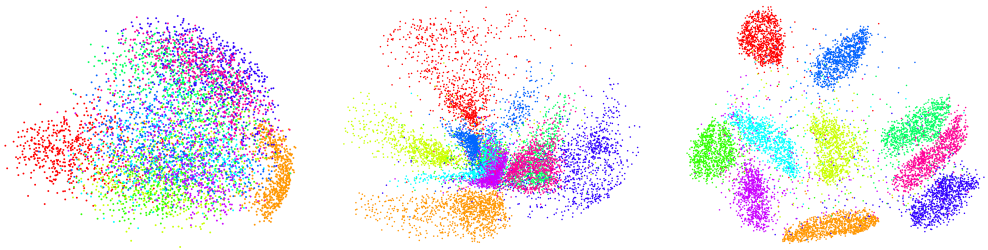
[27] The Moore-Penrose pseudoinverse is usually applied to matrices to solve, e.g., overdetermined linear equation systems. When applied to a non-null vector, the result is the (conjugate) transposed vector which is divided by the squared vector length (or magnitude) of the input vector.

Practical Application and Weaknesses

Many, especially non-linear, feature transforms show very impressive performances on artificial datasets such as the Swiss roll. However, the properties of real-world datasets often complicate the practical application and it is possible that the generated features are less useful for machine learning than the low-level input data. The results of experiments on real-world datasets in [Van der Maaten et al., 2009] prove that most of the feature transform algorithms do not lead to an improved classification performance. This can be explained with the following list of possible weaknesses:

- *No-free-lunch theorem:* There are numerous algorithms proposed for feature transforms based on completely different models. There is likely no best feature transform algorithm that optimally performs for all datasets.
- *Estimation of the target dimensionality:* The estimation of the intrinsic dimensionality of the manifold can fail and an, e.g., unsuitably low-dimensional representation can be generated that loses important information.
- *Suboptimal hyperparameters:* Like classifiers, many manifold learning algorithms have specific hyperparameters that have a huge impact on the resulting transform. The use of suboptimal hyperparameter values can tremendously degrade the performance.
- *Sensitivity to noise and outliers:* Many algorithms are highly influenced by violations of the smooth manifold assumptions.
- *Numerical problems during optimization:* As the data is noisy, e.g., ill-conditioned eigenvector problems can occur that lead to unstable or even unusable solutions.
- *Curse of dimensionality:* Ironically, feature transforms with the goal of dimensionality reduction can also suffer from negative effects of high-dimensional feature spaces as they also incorporate, e.g., Euclidean distance metrics.

Even though the list of problems is rather long, the usefulness of manifold learning and feature transforms for machine learning tasks is reported, e.g., in [Kim et al., 2005] and [Fukumizu et al., 2004]. Figure 3.15 presents quite impressive results for the popular MNIST dataset [LeCun and Cortes, 2010] for handwriting recognition. This dataset contains 28×28 pixel grayscale images of handwritten digits (0, 1, ..., 9). When this image data is used as feature vector, a 784 dimensional feature space is obtained. In this example, three feature transforms, namely PCA, an Autoencoder and the parametric stochastic neighborhood embedding (t-SNE) [Van der Maaten, 2009], are



(a) PCA (b) Autoencoder (c) t-SNE (parametric)

Figure 3.15: Application of three feature transformations on 10,000 images from the MNIST dataset, taken from [Van der Maaten, 2009]. The data is transformed from 784 into two dimensions. Each point represents a digit while there is a different color for each class.

used to visualize 10,000 images in two dimensions. The popular PCA fails to produce a reasonable representation as the class regions are strongly overlapping; also the Autoencoder shows partly overlapping regions. The parametric t-SNE transform produces a well usable representation with nicely separated clusters. Note that all of these feature transforms are unsupervised and a representation as the one in figure 3.15 (c) is well suited even for rather simple classifiers.

Comparison to Visual Object Recognition and Kernel Methods

An interesting observation is that the concept of manifold learning has some relations to the manifold untangling hypothesis in the brain discussed in section 3.3.1. There is evidence that a process of flattening and unfolding highly tangled manifolds into simpler representations takes place in the brain (see figure 3.10). This principally resembles the unfolding process of manifold learning, which is shown in figure 3.11. Additionally, the dimensionalities are reduced in both cases. However, the main differences are the tremendously higher dimensionalities of the neuronal populations in the brain.

The concept of dimensionality reduction seems to be contrary to the idea of kernel methods [Vert et al., 2004] that are successfully applied in, e.g., the SVM (see section 2.1.2). Most kernel methods implicitly transform the feature data into a higher-dimensional space – Gaussian kernels even use feature spaces of infinite dimensionality[28]. The idea is that inside of these high-dimensional spaces the classification problem becomes linearly separable. It is worth noting that some manifold learning algorithms internally use ker-

[28]The Gaussian kernel has a dimensionality of infinity because it contains the exponential function (see equation 2.15 in section 2.1.2) – the Taylor series expansion of $\exp(x)$ contains infinitely many items, e.g., at $x_0 = 0$ the expansion equals $\sum_{n=0}^{\infty} \frac{x^n}{n!}$. However, due to the kernel trick the higher-dimensional feature space never needs to be calculated explicitly.

Challenges \ Solutions	Cross-validation	Learning algorithm selection	Hyperparameter optimization	Feature selection	Feature preprocessing	Representation Learning
Curse of dimensionality		*		**	*	**
– general negative effects		*		*	**	**
– noisy or irrelevant features		*	*	**	*	*
– small training sample sizes	*	*		*		*
Bad generalization due to bias-variance dilemma, overfitting	**	**	**	*		*
Bad generalization due to the representational bias	**					
Suboptimal learning algorithm	**	**				
Suboptimal hyperparameters	**		**			

Table 3.1: Matrix of central machine learning challenges with corresponding solutions. The meaning of the scale is: ** = very helpful, * = potentially helpful.

nel methods, e.g., Kernel-PCA [Schölkopf et al., 1998] or Kernel-LDA [Mika et al., 1999], but the target feature space is lower-dimensional than the input space.

3.4 Discussion

This chapter presents two aspects of solutions for machine learning challenges. First, in section 3.2 a portfolio of established solutions is presented that tackle different parts of the challenges. This "classical" portfolio contains cross-validation, algorithm selection, hyperparameter optimization, feature selection and preprocessing. Secondly, the field representation learning is motivated and discussed in section 3.3. Automatic feature construction techniques are applied to learn better features out of low-level data. A unifying feature transform interface is introduced that handles any manifold learning, feature data transform and dimensionality reduction method.

3.4.1 Comparison of Challenges and Solutions

Table 3.1 shows a matrix of machine learning challenges and potential solutions for each aspect. Model selection methods, consisting of algorithm selection and hyperparameter optimization, are helpful in resolving gener-

alization issues. These are caused either due to the selection of unsuitable features, algorithms or an overfitting to the training dataset. Cross-validation should be used in every case to estimate the generalization performance. This prevents the selection of such unsuitable algorithm and hyperparameter combinations. Furthermore, cross-validation also helps to select useful features to cope with generalization issues caused by the representational bias.

The curse of dimensionality contains several aspects that are most effectively solved with feature selection, preprocessing and representation learning. Feature transforms, one aspect of representation learning, can principally be more effective in dimensionality reduction than feature selection as a transform extracts information from all dimensions and compresses it. Consider the example in figure 3.15 in which the 784 dimensional image data is transformed into two most relevant dimensions – a selection of two single features from the raw pixel data will likely never be that useful. On the other hand, especially non-linear feature transforms can be sensitive to noisy features and fail.

Additionally, algorithm selection can also be helpful to deal with the curse of dimensionality when the classifier portfolio contains methods with internal feature relevance consideration such as random forests. Furthermore, hyperparameter optimization can be useful with noisy features when the smoothness of the decision boundary can be tuned, e.g., with the kernel width γ_{Gauss} for the SVM or the number of neighbors N_{Neigh} in the kNN classifier.

3.4.2 Motivation for a Holistic Solution

Even though powerful solutions and methods have been developed and established, the overview in table 3.1 clearly shows that no solution alone is a remedy against all challenges at the same time. The challenges can occur in any combination or even altogether. This considerably motivates a holistic approach which contains all aforementioned solutions. However, a manual incorporation of all these aspects is even infeasible for experts. Therefore, a fully automated framework is needed to provide the best possible *combination* of machine learning solutions within a reasonable amount of time.

Chapter 4

The AROMS-Framework

This chapter proposes the *Automatic Representation Optimization and Model Selection Framework*, abbreviated as AROMS-Framework, which is the main contribution of this work. It is basically a holistic framework that automatically configures suitable algorithm combinations for classification tasks. For the sake of clarity the whole AROMS-Framework is subdivided into several chapters, i.e. chapters 4 – 6. In this chapter the general concept is motivated and the central structure, the *classification pipeline*, is presented.

This chapter is organized as follows. In section 4.1 the requirements that lead to the main design principles of the framework concept are analyzed. An overview of the proposed AROMS-Framework is given in section 4.2. The classification pipeline is introduced in section 4.3 and its input data is discussed in section 4.4. The pipeline elements along with their motivation and functionality are described in section 4.5. The classification pipeline is adapted using a set of variables denoted as the *pipeline configuration*, which is discussed in section 4.6. Finally, section 4.7 provides an overview of the implementation of the AROMS-Framework.

4.1 Framework Requirements and Concept

Every machine learning task is individual regarding its feature distribution, class structure and generalization problem. A holistic approach to optimize machine learning has to deal with all of the typical challenges (see section 2.2), which can be summarized as:

- the curse of dimensionality arising from high-dimensional feature spaces, few training samples and noisy features,
- the nonexistent general purpose machine learning algorithm with optimal hyperparameters for every task (no-free-lunch theorem),

- the trade-off between generalization and overfitting, regarding the algorithm tuning, but also the feature selection due to the representational bias (see section 2.2.2).

The proposed AROMS-Framework comprises a fusion of the "classical" solutions, discussed in section 3.2, as well as methods from the field of representation learning, presented in section 3.3. Three main principles can be extracted from this large amount of solutions, namely

- dimensionality reduction,
- automatic algorithm portfolio selection and
- automatic hyperparameter optimization.

These principles are the foundation of the general design concept of the AROMS-Framework which are motivated and explained in the following.

4.1.1 Dimensionality Reduction

One important principle is dimensionality reduction. Its aim is to achieve a more suitable representation without losing important information. This aspect is also biologically motivated (see section 3.3.1) and can be subdivided into two parts.

First, the removal of irrelevant information is crucial for well generalizing machine learning algorithms. Features that contain relevant information may also be noisy, but a classifier model can potentially make use of the information. However, in combination with few training samples the chance of averaging out the noise decreases and the risk of overfitting to a wrong model based on noise increases. The principle of neglecting irrelevant information can also be found in the preattentive stage of the human vision system that first focuses on important areas in the field of vision.

The second aspect in dimensionality reduction are feature transforms based on representation learning techniques. Rather than simply removing dimensions, these feature transforms have the potential to find a usually lower-dimensional representation by analyzing the, e.g., geometrical properties of the feature distribution. With a suitable feature transform complex feature distributions can be simplified so that, e.g., linear classifiers perform well. This principle resembles the "manifold untangling" theory in the brain (see section 3.3.1) when performing object recognition. Evidence is found that the brain uses multiple, consecutive neural representation transforms to achieve its extraordinary recognition abilities.

4.1.2 Automatic Portfolio-based Algorithm Selection

The no-free-lunch theorem states that there is no general purpose machine learning algorithm. Some algorithms such as support vector machines (SVM)

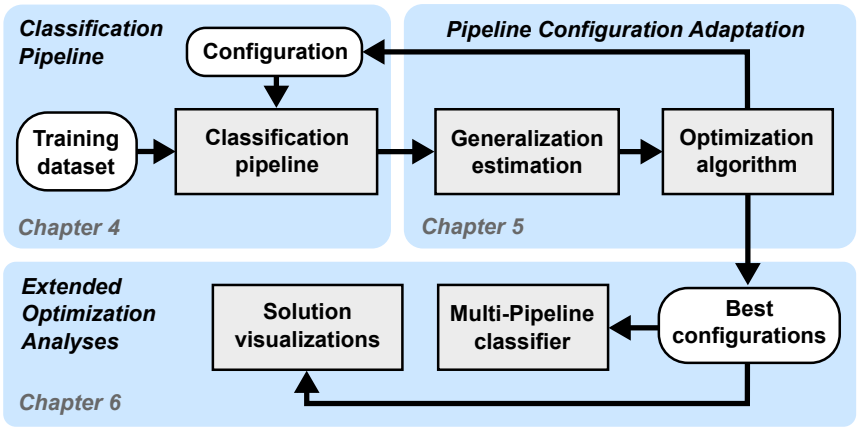


Figure 4.1: Overview of the proposed AROMS-Framework and its subdivision into three chapters.

show an excellent performance on a wide range of classification tasks. However, there is no straightforward way to predict which classifier performs best for a given task. The same aspect is true for feature transforms and also preprocessing algorithms. So the framework needs to contain portfolios of algorithm alternatives for each processing stage. The selection of these algorithms needs to be fully automatic so that the final performance does not depend on the knowledge or time budget of the user of the framework.

4.1.3 Automatic Hyperparameter Optimization

Most machine learning algorithms can be controlled by a set of different hyperparameters. These hyperparameters have a great effect on the learning behavior and need to be tuned for each learning task to achieve the best performing algorithm. Thus, all hyperparameters of the involved algorithms should be optimized automatically. The need for manual work has to be minimized as much as possible.

4.2 Overview of the AROMS-Framework

The AROMS-Framework has to include numerous aspects and requirements into a computational framework. In order to be able to present all aspects in a reasonable way, the framework is subdivided into different parts and three chapters. An overview of these parts of the framework can be found in figure 4.1.

The classification pipeline is the core structure of the whole framework and is described in section 4.3 within this chapter. It contains all machine learning algorithms and other processing steps. The configuration of the clas-

sification pipeline allows the adaptation of the pipeline's degrees of freedom consisting of feature selection, algorithm and hyperparameter determination. The configuration can be understood as a global hyperparameter set for the pipeline.

The automatic adaptation of the pipeline configuration to a given learning task is an optimization problem which is discussed in chapter 5. First, a suitable optimization metric has to be determined that estimates the generalization performance of the classification pipeline with all its components. The actual optimization algorithm uses this metric to adapt the configuration in a feedback loop with the classification pipeline and the training dataset.

Chapter 6 presents extended optimization analyses that are based on the results of the optimization algorithm. Usually, many reasonably well working configurations are evaluated to find the best one. The goal of these extensions is to make use of these results. One application is the construction of multi-pipeline classifiers with improved generalization capabilities. Additionally, suitable visualization techniques are discussed that investigate the statistical properties of the configuration distribution. This allows a gain of knowledge about the classification problem and its solutions.

4.3 Classification Pipeline

This section as well as the following ones describe the classification pipeline structure. The term "pipeline" is used because it is based on the *pipes and filters* design pattern introduced by [Buschmann et al., 1996]. This pattern is applicable for systems that process streams of data with several consecutive processing steps that are encapsulated in so-called filters. The data is passed to the next filter through a pipe and the whole structure resembles a pipeline with feedforward processing. Figure 4.2 shows the general structure of the classification pipeline and its four pipeline elements, i.e.

1. the feature selection element $E_{FeatSel}$,
2. the feature preprocessing element $E_{PreProc}$,
3. the feature transform element E_{Trans} and
4. the classifier element $E_{Classifier}$.

A classification pipeline is formally defined as the set of pipeline elements

$$\Theta = \{E_{FeatSel}, E_{PreProc}, E_{Trans}, E_{Classifier}\}\,. \tag{4.1}$$

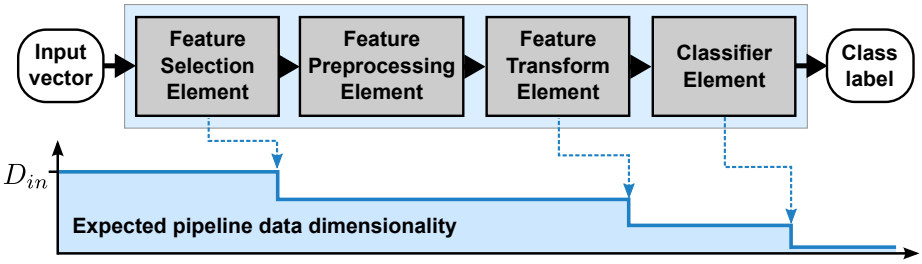


Figure 4.2: Classification pipeline with four pipeline elements. The typical dimensionality of the processed data is shown in the plot at the bottom for each pipeline element. Note that the feature preprocessing element does not reduce the dimensionality and that the output class label is one-dimensional.

4.3.1 General Processing Overview

At first, the feature selection element should select a subset of relevant features and thus removes irrelevant ones that could potentially disturb all further algorithms. This is the first step of dimensionality reduction within the framework. Secondly, the feature preprocessing element performs rather simple processing on the feature vectors with the aim to optimize the data for machine learning. Thirdly, the feature transform element applies the feature transform, which is the second dimensionality reduction step in the framework. Finally, the classifier element performs the classification based on the processed feature data – after feature selection, preprocessing and transform.

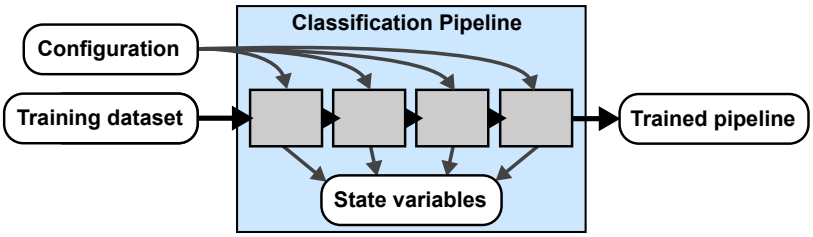
The name of the proposed framework, Automatic Representation Optimization and Model Selection Framework (AROMS), is directly related to the functionality of the classification pipeline: First, the aspect of *representation optimization* is covered by the first three pipeline elements as the feature representation is changed by means of feature selection, preprocessing and transform. Secondly, the aspect of *model selection* is included in multiple ways as well. The "classical" classifier models are certainly situated within the classifier element. However, also the feature preprocessing and feature transform algorithms also contain model functions that are selected and adapted. Lastly, the aspect of *automatism* is covered with the automatic adaptation of the pipeline configuration (see chapter 5).

4.3.2 Classification Pipeline Modes

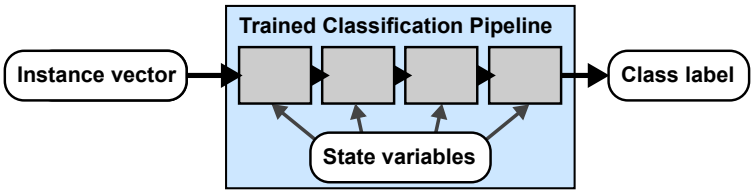
The classification pipeline offers two modes, namely the *training mode* and the *classification mode*, which are depicted in figure 4.3.

First, the training mode is needed to set up the pipeline for classification

$$\Theta = f_{TrainPipeline}(\theta, T_{Train}) \tag{4.2}$$



(a) Training mode



(b) Classification mode

Figure 4.3: Overview of the two modes of the classification pipeline. In the training mode (a) the pipeline's algorithms are trained and their internal model parameters are stored in the state variables. In the classification mode (b) the trained pipeline is used to assign a class label to an input instance vector.

in which the input is a pipeline configuration θ and a training dataset T_{Train}. The configuration controls the pipeline's components and hyperparameters that influence the learning behavior – it is fully defined later in section 4.6. In this mode each pipeline element adapts its models and internal model parameters to the training dataset. These model parameters, such as network weights, are stored in the state variables.

In the classification mode a trained pipeline Θ is used to classify previously unseen instance vectors $\mathbf{x}$ with the function

$$y = f_{ProcessPipeline}(\Theta, \mathbf{x}) \in S_{Classes}\,. \tag{4.3}$$

The feature vector $\mathbf{x}$ is processed by the pipeline elements that use the models stored in the state variables. The plot in figure 4.2 shows the expected two-stage dimensionality reduction process of the feature space – in the feature selection and feature transform element. Finally, a one-dimensional class label y is the output of the pipeline.

The precise functionality of each pipeline element in the training and classification mode is explained in particular within the sections 4.5.1 – 4.5.4.

4.4 Input Data

Before the components of the classification pipeline can be described in detail, the input data for the AROMS-framework has to be defined. The "classical" definition of the supervised classification problem can be found in section 2.1. It is based on labeled feature vectors out of an D_{in}-dimensional feature space. However, in practical applications different kinds of feature sources, e.g., multiple sensors and multiple feature descriptors are combined. These feature sources need not necessarily be one-dimensional and therefore, the concept of *feature groups* is introduced. The data format for the AROMS-Framework consists of a set of N_{Groups} feature groups

$$S_{FeatGroups} = \{featGroup(1), featGroup(2), \ldots, featGroup(N_{Groups})\} \tag{4.4}$$

in which

$$featGroup(l) = (\mathbf{x}_{Group,l}, D_{Group,l}, R_{Group,l}) \tag{4.5}$$

is a tuple of a sub feature vector $\mathbf{x}_{Group,l} \in \mathbb{R}^{D_{Group,l}}$ and a description text $R_{Group,l}$ with the feature name. All sub feature vectors are concatenated to obtain the input feature vector

$$\mathbf{x}_{in} = \left[\mathbf{x}_{Group,1}, \mathbf{x}_{Group,2}, \ldots, \mathbf{x}_{Group,N_{Groups}}\right] \tag{4.6}$$

having a total dimensionality of the sum of sub dimensionalities

$$D_{in} = \sum_{l=1}^{N_{Groups}} D_{Group,l}\,. \tag{4.7}$$

The connection between the feature group index and the component index of $\mathbf{x}_{in}$ has to be stored for the statistical analyses of the extended optimization analyses (see chapter 6).

To give an example for this data structure, consider a typical image-based classification task with three feature groups

- the object area in pixels (one dimensional),
- the mean gray value of the object (one dimensional),
- an LBP descriptor (256 dimensional, see section 2.2.1).

This leads to a feature group set with $N_{Groups} = 3$ for an example instance object of

$$\begin{aligned} S_{FeatGroups} = \{&(\mathbf{x}_{Group,1} = [150],\ D_{Group,1} = 1,\ R_{Group,1} = \text{`}object\ size\text{'}),\\ &(\mathbf{x}_{Group,2} = [200.5],\ D_{Group,2} = 1,\ R_{Group,2} = \text{`}mean\ gray\ value\text{'}),\\ &(\mathbf{x}_{Group,3} = [3, 20, 7, \ldots],\ D_{Group,3} = 256,\ R_{Group,3} = \text{`}LBP\ descriptor\text{'})\} \end{aligned} \tag{4.8}$$

with a total dimensionality of $D_{in} = 1 + 1 + 256 = 258$.

4.5 Pipeline Elements

The classification pipeline consists of four pipeline elements, namely the feature selection, feature preprocessing, feature transform and classifier element, whose functionality is presented in the following subsections.

4.5.1 Feature Selection Element

The first pipeline element is the feature selection element $E_{FeatSel}$, which is responsible for the first step of dimensionality reduction in the AROMS-Framework. The feature selection is realized by the determination of a set of element indices of the concatenated feature vector $\mathbf{x}_{in} \in \mathbb{R}^{D_{in}}$. The set of all possible subsets depends on the dimensionality D_{in} of the feature vectors:

$$S_{AllFeatCombi} = \mathcal{P}(\{1, 2, \ldots, D_{in}\}) \setminus \emptyset\,. \tag{4.9}$$

The empty subset is excluded as there has to be at least one feature to allow a classification. The total number of subsets is given by

$$|S_{AllFeatCombi}| = 2^{D_{in}} - 1 \tag{4.10}$$

elements. For example, if $D_{in} = 3$ then seven index subsets are possible, namely

$$S_{AllFeatCombi} = \{\{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}\,. \tag{4.11}$$

Training Mode

In the training mode a feature index subset

$$S_{FeatSubSet} = \{j_1, j_2, \ldots, j_{D_{FeatSel}}\} \in S_{AllFeatCombi} \tag{4.12}$$

is determined leading to a dimensionality of

$$D_{FeatSel} = |S_{FeatSubSet}| \leq D_{in}. \tag{4.13}$$

Clearly, the dimensionality of the feature space is reduced in all cases in which the full feature set *not* is selected:

$$D_{FeatSel} < D_{in} \quad \Leftrightarrow \quad S_{FeatSubSet} \neq \{1, 2, \ldots, D_{in}\}\,. \tag{4.14}$$

The choice of this subset has a great impact on the classification performance of the whole pipeline and the number of possible combinations grows exponentially with the number of input dimensions D_{in} (see equation 4.10). The feature subset $S_{FeatSubSet}$ has to be adapted for each learning task and therefore it is included in the pipeline configuration (see section 4.6).

Once a feature subset $S_{FeatSubSet}$ is selected, it is stored in the state variables. The feature subset selection is performed by concatenating the selected feature dimensions

$$\mathbf{x}_{FeatSel}^{(i)} = \left[x_{j_1}^{(i)}, x_{j_2}^{(i)}, \ldots, x_{j_{D_{FeatSel}}}^{(i)}\right] \in \mathbb{R}^{D_{FeatSel}} \tag{4.15}$$

in which the elements $x_j^{(i)}$ originate from the full input feature vector $\mathbf{x}_{in}$. All feature vectors of the training dataset are processed and a new training dataset

$$T_{FeatSel} = \left\{\left(\mathbf{x}_{FeatSel}^{(i)}, y^{(i)}\right)\right\}. \tag{4.16}$$

with $1 \leq i \leq N_T$ is obtained to be passed to the next pipeline element.

Classification Mode

In the classification mode a feature vector $\mathbf{x} \in \mathbb{R}^{D_{in}}$ is passed to $E_{FeatSel}$. The determined feature subset $S_{FeatSubSet}$ is taken from the state variables and the feature subset selection in equation 4.15 is performed. The output is a feature vector $\mathbf{x}_{FeatSel} \in \mathbb{R}^{D_{FeatSel}}$.

4.5.2 Feature Preprocessing Element

The second pipeline element is the feature preprocessing element $E_{PreProc}$ that optimizes the feature data for any further machine learning algorithm. Relatively simple preprocessing steps such as, e.g., feature scaling (see section 3.2.3) can effectively improve the classification performance in cases of different value ranges of the features. However, it is not obvious which preprocessing method leads to the best classification performance. Therefore, the design principle of portfolio-based algorithm selection is employed in $E_{PreProc}$ with a portfolio set of $N_{PreProc}$ preprocessing methods

$$S_{PreProc} = \{A_{PreProc,1}, A_{PreProc,2}, \ldots, A_{PreProc,N_{PreProc}}\} \tag{4.17}$$

that fit to the feature preprocessing interface $A_{PreProc}$ described in section 3.2.3. The feature preprocessing algorithms are listed in appendix B. Even though feature preprocessing usually improves the generalization, it might be the case that the unprocessed data works even better. Therefore, the identity function – or *no preprocessing* – is also included in the set $S_{PreProc}$.

Training Mode

In the training mode a preprocessing function $A_{PreProc} \in S_{PreProc}$ has to be selected from the portfolio. This selection is included in the pipeline

configuration (see section 4.6). Once a preprocessing method $A_{PreProc}$ is determined, the corresponding model parameter estimation function

$$S_{Params,PreProc} = f_{PreProc,estimate}(T_{FeatSel}) \tag{4.18}$$

is applied using the training dataset $T_{FeatSel}$ which is received from the previous feature selection element. The result is the set of model parameters $S_{Params,PreProc}$, which contains, e.g., minimum and maximum values for each dimension in case of rescaling. These preprocessing model parameters are stored in the state variables. Feature vectors are processed with the preprocessing function from the selected method $A_{PreProc}$, i.e.

$$\mathbf{x}^{(i)}_{PreProc} = f_{PreProc}(S_{Params,PreProc}, \mathbf{x}^{(i)}_{FeatSel}) \tag{4.19}$$

and put to a new dataset

$$T_{PreProc} = \left\{ \left(\mathbf{x}^{(i)}_{PreProc}, y^{(i)} \right) \right\} \tag{4.20}$$

with $1 \leq i \leq N_T$, which is passed to the next pipeline element. Note that the dimensionality is not changed by the proposed preprocessing methods and thus $\mathbf{x}^{(i)}_{PreProc} \in \mathbb{R}^{D_{FeatSel}}$.

Classification Mode

In the classification mode incoming vectors $\mathbf{x}_{FeatSel}$ from the feature selection element are processed with the selected preprocessing function according to equation 4.19 and passed to the next pipeline element as $\mathbf{x}_{PreProc}$.

4.5.3 Feature Transform Element

The feature transform element E_{Trans} is the third pipeline element and it is responsible for the second stage of dimensionality reduction in the AROMS-Framework. The E_{Trans} pipeline element uses representation learning methods based on the *feature transform interface* which is presented in section 3.3.2. The potential benefits of feature transforms are two-fold: First, most transforms reduce the dimensionality of the feature space and thus help to circumvent the curse of dimensionality. Secondly, the feature representation can be improved so that simpler classifier models perform better. At best, both benefits can be achieved with the same feature transform.

The design principle of portfolio-based algorithm selection is considered with a portfolio of N_{Trans} feature transform interfaces

$$S_{Trans} = \{A_{Trans,1}, A_{Trans,2}, \ldots, A_{Trans,N_{Trans}}\}. \tag{4.21}$$

The feature transforms in this portfolio are listed in appendix C. Even though feature transforms have the potential to improve the classification process, it might be the case that no transform is able to build a suitable model. Therefore, the identity function – or *no transform* – is also included in the set S_{Trans}. In this special case, no dimensionality reduction is performed.

Training Mode

In the training mode a specific feature transform interface $A_{Trans} \in S_{Trans}$ needs to be chosen. The training process requires the target dimensionality D_{low}, which is determined with $D_{low} = D_{Trans} \in \mathbb{N}$. The previous pipeline element $E_{PreProc}$ passes feature vectors with a dimensionality of $D_{FeatSel}$. Therefore, the target dimensionality needs to fulfill the constraints of $1 \leq D_{Trans} \leq D_{FeatSel}$ as the dimensionality cannot be increased. Finally, the corresponding hyperparameter values $S_{\mathbb{H},A_{Trans}}$ (see section 3.2.4) of the feature transform have to be selected. The selection and parameterization of the feature transform interface are included in the pipeline configuration (see section 4.6). After that, the selected feature transform is trained by the corresponding manifold learning function in A_{Trans}, i.e.

$$S_{TransParams} = f_{LearnTrans}(T_{PreProc}, D_{Trans}, S_{\mathbb{H},A_{Trans}}) \tag{4.22}$$

using the training dataset $T_{PreProc}$ that is passed from the previous feature preprocessing element. The result is the set of learned model parameters $S_{TransParams}$, which is stored in the state variables of E_{Trans}. The model is used to entirely transform the training dataset $T_{PreProc}$ for the next and last pipeline element – the classifier: Each vector is transformed by the feature transform function that corresponds to the selected method A_{Trans}, i.e.

$$\mathbf{x}^{(i)}_{Trans} = f_{Transform}(S_{TransParams}, \mathbf{x}^{(i)}_{PreProc})\,. \tag{4.23}$$

The dimensionality of the feature vectors is in most cases reduced to $\mathbf{x}^{(i)}_{Trans} \in \mathbb{R}^{D_{Trans}}$. Note that some feature transforms, e.g., LDA and LMNN (see appendix C), ignore the target dimensionality D_{Trans}. In this case, the resulting output dimensionality of the transform is used. The resulting dataset contains all transformed vectors with

$$T_{Trans} = \left\{ \left(\mathbf{x}^{(i)}_{Trans}, y^{(i)} \right) \right\} \tag{4.24}$$

and $1 \leq i \leq N_T$.

A general problem of feature transforms in real-world applications is that the resulting lower-dimensional features have no guarantee to be numerically stable. Especially non-linear transforms can become unstable and produce extremely large or small values when the hyperparameters are unsuitable. Experiments have shown that values between $\pm 10^{200}$ can occur frequently, even though the previous pipeline element applies a preprocessing method like rescaling on the data. Such unstable features can likely disturb the following classifier algorithms and even cause frequent program crashes of normally stable program libraries. In order to prevent these problems, a rescaling of the transformed features to a range of $[0, 1] \subset \mathbb{R}$ is performed when a feature transform other than the identity is chosen. The actual rescaling is similar

to the rescaling in the feature preprocessing element (see section 4.5.2). The transformed dataset is used to estimate the minimum and maximum values of each new feature dimension

$$S_{Params,Rescaling} = f_{Rescaling,estimate}(T_{Trans}), \tag{4.25}$$

which are needed to rescale the data with

$$\mathbf{x}^{(i)}_{Trans'} = f_{PreProc}(S_{Params,Rescaling}, \mathbf{x}^{(i)}_{Trans}) \,. \tag{4.26}$$

Finally, the transformed and rescaled output training dataset is defined as

$$T_{Trans'} = \left\{ \left(\mathbf{x}^{(i)}_{Trans'}, y^{(i)} \right) \right\} \tag{4.27}$$

for $1 \leq i \leq N_T$ and it is passed to the classifier pipeline element.

Classification Mode

In the classification mode incoming feature vectors of the preprocessing element are transformed using the transform function and the stored model parameter set in

$$\mathbf{x}_{Trans} = f_{Transform}(S_{TransParams}, \mathbf{x}_{PreProc}). \tag{4.28}$$

After that, if a feature transform other than the identity is chosen, the features are rescaled using

$$\mathbf{x}_{Trans'} = f_{PreProc}(S_{Params,Rescaling}, \mathbf{x}_{Trans}) \tag{4.29}$$

and are passed to the classifier pipeline element.

4.5.4 Classifier Element

The final pipeline element is the classifier element $E_{Classifier}$, which predicts class labels for feature vectors. A suitable classifier has to be chosen to work with the feature vectors that have been processed by the previous pipeline elements. There are many classifier concepts, which are discussed in section 2.1.2, and a unifying interface is needed to handle them conveniently within the AROMS-Framework. The classifier interface is defined as a tuple

$$A_{Class} = (f_{trainClass}, S_{\mathbb{H},A_{Class}}, S_{Model,Class}, f_{Classifier}) \tag{4.30}$$

consisting of a training function $f_{trainClass}$, a set of classifier specific hyperparameters $S_{\mathbb{H},A_{Class}}$ (see section 3.2.4), the resulting classifier model $S_{Model,Class}$ and a classification function $f_{Classifier}$. The classifier training process builds

a model using the training data $T_{Trans'}$ from the feature transform element and the hyperparameter values

$$S_{Model,Class} = f_{trainClass}(T_{Trans'}, S_{\mathbb{H},A_{Class}}). \tag{4.31}$$

The classifier model $S_{Model,Class}$ is a set with arbitrary variables, e.g., network weights in case of artificial neural networks, depending on the classifier concept. The classification function

$$y = f_{Classifier}(S_{Model,Class}, \mathbf{x}_{Trans'}) \in S_{Classes} \tag{4.32}$$

assigns a label y to a feature vector from the feature transform element based on the trained model $S_{Model,Class}$.

The design principle of portfolio-based algorithm selection is incorporated with a portfolio set of $N_{Classifiers}$ classifiers

$$S_{Classifiers} = \{A_{Class,1}, A_{Class,2}, \ldots, A_{Class,N_{Classifiers}}\}. \tag{4.33}$$

Note that each classifier has individual training functions, hyperparameters and classification functions. The classifiers that are used within this work are listed in appendix D.

Training Mode

In the training mode a suitable classifier $A_{Class} \in S_{Classifiers}$ and its hyperparameter values $S_{\mathbb{H},A_{Class}}$ need to be selected, which are included in the pipeline configuration (see section 4.6). This classifier is trained using $f_{trainClass}$ (see equation 4.31) that corresponds to the selected classifier A_{Class}. The training dataset $T_{Trans'}$ is received from the feature transform element. Note that the feature vectors are processed by many algorithms at this point of the pipeline – a feature subset is selected, a preprocessing method and a feature transform are applied. After training, the resulting classifier model $S_{Model,Class}$ is stored in the state variables.

Classification Mode

In the classification mode a feature vector $\mathbf{x}_{Trans'}$ from the feature transform element is received and passed to the trained classifier. The classification function $f_{Classifier}$ that corresponds to the selected classifier A_{Class} and the learned classifier model $S_{Model,Class}$ are used according to equation 4.32. The resulting class label y is the final output of the pipeline and can be used for the desired application or evaluation purposes.

4.6 Pipeline Configuration

The proposed classification pipeline and its four pipeline elements have several degrees of freedom to adapt to a classification task. These variables are summarized in the *pipeline configuration* θ that controls the behavior of the pipeline elements and can be seen as the global hyperparameter of the whole classification pipeline. The pipeline configuration is defined as a tuple

$$\theta = \left(\underbrace{S_{FeatSubSet}}_{\text{feature selection}} , \underbrace{A_{PreProc}}_{\text{preprocessing}} , \underbrace{A_{Trans}, D_{Trans}, S_{\mathbb{H},A_{Trans}}}_{\text{feature transform}}, \underbrace{A_{Class}, S_{\mathbb{H},A_{Class}}}_{\text{classifier}} \right) \tag{4.34}$$

consisting of

- the feature subset $S_{FeatSubSet}$,
- the feature preprocessing method $A_{PreProc}$,
- the feature transform method A_{Trans} and the corresponding
- feature transform hyperparameters $S_{\mathbb{H},A_{Trans}}$ as well as the
- target dimensionality D_{Trans},
- the classifier A_{Class} and the corresponding
- classifier hyperparameter values $S_{\mathbb{H},A_{Class}}$.

The set of all possible pipeline configurations for a learning task is denoted as $S_\theta(T_{Train})$. It depends on the learning task because the set of all possible feature subsets $S_{AllFeatCombi}$ is defined by the dimensionality D_{in} and $S_{FeatSubSet} \in S_{AllFeatCombi}$ (see section 4.5.1).

Consider the following toy example of a pipeline configuration with $D_{in} = 5$ feature dimensions in the training dataset. A valid configuration for the classification pipeline would contain the following items: The feature subset is $S_{FeatSubSet} = \{1, 3\}$ to select the first and the third feature of the concatenated feature groups. The selected feature preprocessing method is rescaling, the feature transform is the Principal Component Analysis (PCA) with a target dimensionality of $D_{Trans} = 1$ and no hyperparameters. The linear SVM classifier is chosen with its hyperparameter value $c_{reg} = 10$.

4.7 Framework Implementation Overview

The AROMS-Framework is implemented in Matlab and its source code is published on *GitHub*[1] in [Bürger, 2016]. Matlab provides a very high level programming language that allows a fast development of complex systems.

[1] *GitHub* is a web-based service for collaborative software development especially for open source projects. The service can be found at `https://github.com`.

However, its computational efficiency is partly lower compared to other programming languages such as C or C++. One of the main reasons for this choice is the availability of the *Matlab Toolbox for Dimensionality Reduction* [Van der Maaten, 2014], which is used in the feature transform pipeline element. The toolbox includes more than 30 different manifold learning and feature transform algorithms (see appendix C). No other programming language or program library contains such a large collection of implementations of these algorithms.

The framework implementation follows object oriented programming concepts. Figure 4.4 shows a UML[2] class and package diagram of the central classes of the framework. The structure of the packages is mainly based on the three parts of the AROMS-Framework, i.e. the classification pipeline, the pipeline configuration adaptation and the extended optimization analyses. However, for implementation reasons, some additional subdivisions are made in the package structure. These are explained in the following.

The first package *FrameworkBase* contains the main classes which control the whole framework. The class *AROMSFrameworkController* receives the input of one or several *ClassificationJob* structures[3] consisting of a classification problem in form of a training dataset. The training dataset is a structure with the following fields:

- The field *dataSetName* contains the name of the dataset in form of a string that is used to generate statistics.
- The field *instanceFeatures* is a structure which contains the feature groups $S_{FeatGroups}$ (see section 4.4) for multiple instances. Every feature group $featGroup(l)$ with $1 \leq l \leq N_{Groups}$ corresponds to one field in the *instanceFeatures* structure whose name string is equal to the description text $R_{Group,l}$ of the feature group. The data of that field stores the actual feature vectors of N_T instances in form of a matrix

$$\mathbf{X}_{Group,l} = \begin{bmatrix} \mathbf{x}_{Group,l}^{(1)} \\ \mathbf{x}_{Group,l}^{(2)} \\ \vdots \\ \mathbf{x}_{Group,l}^{(N_T)} \end{bmatrix} \in \mathbb{R}^{N_T \times D_{Group,l}} \tag{4.35}$$

  using the double-precision floating-point format. The ith row of $\mathbf{X}_{Group,l}$ is the feature vector of the ith instance of the lth feature group. The

[2]UML (Unified Modeling Language) is graphical modeling language to specify software structures. An introduction can be found in comprehensive books such as [Booch et al., 2005].

[3]A structure in Matlab – and also in many other programming environments – contains fields with data that are referenced by a name string. These fields can contain arbitrary kinds of data, e.g., strings, matrices or structures.

feature group dimensionality $D_{Group,l}$ is equal to the number of columns in $\mathbf{X}_{Group,l}$.

- The *targetClasses* field contains the ground truth labels of the instance vectors. They are represented in form of a column vector

$$\mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N_T)} \end{bmatrix} \in \mathbb{N}^{N_T \times 1} \tag{4.36}$$

  in which the ith row is the numeric class label index $y^{(i)}$ of the ith instance. Note that the *targetClasses* field is only required for training and test datasets in which the ground truth labels are needed for the pipeline training and evaluation. If new instances should be classified, the field can be empty.

- The *classNames* field contains a list with the names of the $N_{Classes}$ classes in form of strings. The class names are useful to generate well understandable statistics.

Furthermore, each *ClassificationJob* also contains specific metaparameters for the AROMS-Framework (see appendix E) that influence, e.g., the optimization algorithms. Most of these metaparameters are needed for testing and evaluating parts of the AROMS-Framework itself (see chapter 7). End users do not need to pass any AROMS-Framework metaparameters; in this case, the standard metaparameters, listed in appendix E, are used. The validity of the metaparameters are checked by the *FrameworkParameterController*, which also fills missing values.

The *FrameworkPipeline* package contains all classes that are needed for the classification pipeline. The abstract class *PipelineClass* comprises an arbitrary number of abstract pipeline elements of type *PipelineElement*. This software structure theoretically allows to handle any kind of data processing pipeline. The *ClassificationPipeline* is a subclass of the *PipelineClass* and owns the four pipeline element objects, i.e. *FeatureSelectionElement*, *FeaturePreProcessingElement*, *FeatureTransformElement* and *ClassifierElement*. These are subclasses of the abstract class *PipelineElement* because they share the same interface to receive and pass data inside of the pipeline. The only differences between the subclasses are the specific functionalities to process the data.

The *FrameworkOptimization* package contains all classes that are related to the pipeline configuration adaptation (see chapter 5). The *FrameworkOptimizationController* controls an instance of an optimization algorithm, which inherits from the abstract class *OptimizationStrategy*. The subclass *StrategyEvolutionary* contains the Evolutionary Algorithm that is presented in

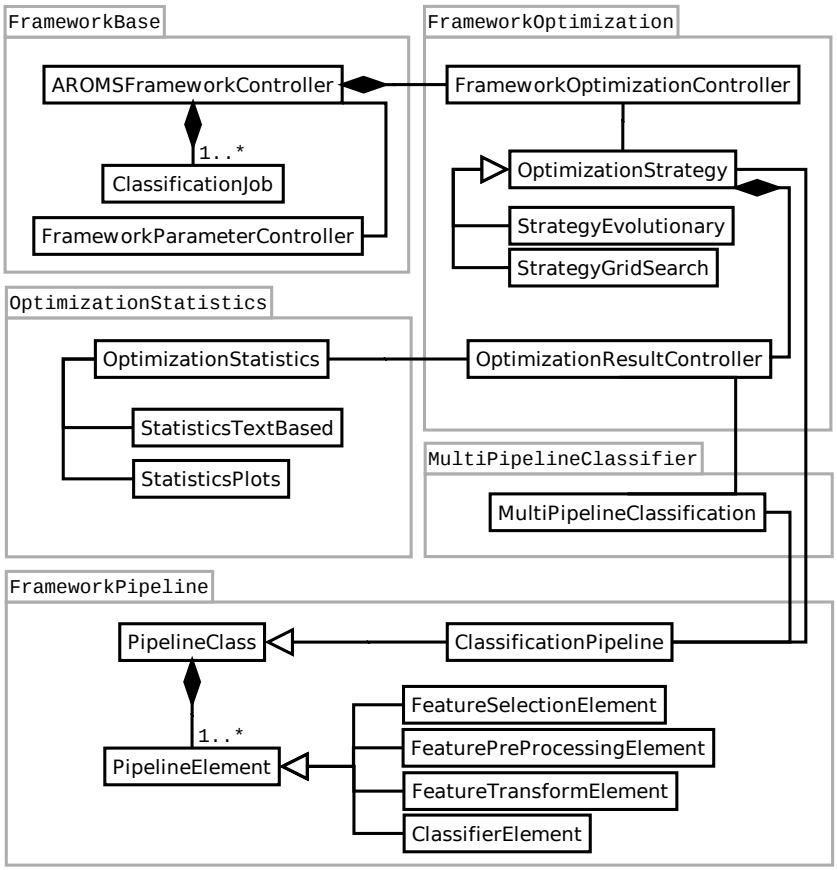


Figure 4.4: Simplified UML class diagram presenting the central parts of the implementation of the AROMS-Framework. Lines without a special symbol denote an *association* of the classes. The filled diamond symbols denote a *composition* relationship and the triangles denote a class *generalization*.

chapter 5. Other optimization strategies can be easily implemented as a subclass of *OptimizationStrategy*. A baseline grid search approach is implemented in the subclass *StrategyGridSearch* that is used as a comparison benchmark in the evaluations (see chapter 7). The results of the optimization process itself are handled by the *OptimizationResultController* class. All results like the accuracy and configuration trajectory as well as time evaluations are stored by this class.

The extended optimization analyses, presented in chapter 6, are divided into two packages. First, the *OptimizationStatistics* package contains all classes that use the results of the optimization process provided by the class *OptimizationResultController* for the generation of statistics. Text-based and graphical statistics are provided which are implemented in the classes *StatisticsTextBased* and *StatisticsPlots*, respectively. The system can automatically generate comprehensive statistics for multiple classification jobs with

different datasets and AROMS-Framework metaparameters. An example are two-dimensional tables in which the datasets are listed in rows whereas metaparameter values are listed in columns. The data part of the table contains specific performance values for each dataset/parameter combination, e.g., accuracy values with standard deviations from the corresponding experiment repetitions. The tables and graphical statistics are automatically exported in formats that can directly be used for scientific publications, e.g., LaTeX[4] and PDF[5] files.

Secondly, the multi-pipeline classifier is realized in the *MultiPipelineClassifier* package using the *MultiPipelineClassification* class. This class also uses the optimization trajectory information provided by the *OptimizationResultController* class. Multiple objects of class *ClassificationPipeline* are generated and used for the multi-pipeline classifier.

[4]LaTeX is a platform independent word processing system that is widely used in academic fields.

[5]The Portable Document Format (PDF) is a platform independent file format which is also suitable for graphics.

Chapter 5

Pipeline Configuration Adaptation

The classification pipeline of the AROMS-Framework, presented in the previous chapter, contains a highly adaptable set of machine learning algorithms for arbitrary classification tasks. The pipeline configuration allows the selection of the feature set, altogether three algorithms and their corresponding hyperparameters. However, the large degree of adaptability comes with the challenge to find a good configuration for a given task within a reasonable time. The problem is that on the one hand, there is no straightforward way to guess the best configuration directly and on the other hand, it is obvious that a brute force approach – trying all possible combinations – is infeasible.

Machine learning experts are faced with exactly the same problem when designing a classification system. Obviously, a naïve approach like choosing a well-known standard classifier, e.g., the Support Vector Machine (SVM), and leaving all hyperparameters at their standard values will likely give a suboptimal result. A more sophisticated methodology would be to tune the most important hyperparameters with grid search, which might already significantly improve the performance. However, grid search is not efficient enough to optimize a large number of hyperparameters in combination with multiple algorithms.

This chapter discusses suitable approaches to adapt the pipeline configuration automatically to a given learning task and it is organized as follows. First of all, the generalization performance of a given configuration needs to be estimated in order to rank them by their "quality" for a specific task. This is done with an adapted version of cross-validation which is presented in section 5.1. Section 5.2 defines the configuration adaptation problem as an optimization problem and analyzes its properties. Section 5.3 discusses suitable algorithms to tackle this optimization problem. Evolution Strategies, a variant of Evolutionary Algorithms, turn out to be suitable for such difficult optimization tasks. However, certain extensions of the standard Evolution

Strategies need to be developed which are described in section 5.4. Based on this approach an optimization algorithm for the configuration adaptation problem is presented in section 5.5. Finally, section 5.6 proposes several variants of the basic optimization algorithm to investigate the interplay of the pipeline components.

5.1 Pipeline Generalization Estimation

The automatic selection of a good configuration leading to a well generalizing classification pipeline is one of the main goals of the AROMS-Framework. The first question is whether a filter or wrapper approach (see section 3.2.2) should be chosen to achieve this aim.

Of course, a *filter approach* which directly suggests a good configuration by simply analyzing statistical properties of the feature distribution in the dataset T_{Train}, would be tempting. However, existing filter approaches are limited to the feature selection problem. And even this aspect is problematic as the interplay of the algorithms in the classification pipeline is complex. The usefulness of a single feature is very hard to predict: A feature dimension may be redundant and therefore a filter approach would remove it. But this particular dimension might have been useful to improve the estimation of a manifold learning algorithm in the feature transform pipeline element. Any further prediction of suitable algorithms for a given dataset T_{Train} using a filter approach seems to be infeasible.

Consequently, a *wrapper approach* is needed that trains multiple classification pipelines and evaluates the actual performance. The aspect of model validation is of central interest here. The actual model is the trained classification pipeline $\Theta = f_{TrainPipeline}(\theta, T_{Train})$ for a given training dataset T_{Train} and a configuration θ. The goal is to predict the generalization performance of the classification pipeline Θ on the test dataset T_{Test} without actually involving the test dataset itself. This can be achieved with suitable generalization estimators $g_{Obj,Gen}(\theta, T_{Train}) \mapsto \mathbb{R}^+$, which are discussed in the following.

5.1.1 Standard Cross-Validation

Cross-validation is a common way to estimate the generalization of classifiers (see section 3.2.1). The standard variant is usually used to estimate the average accuracy of a single classifier on the validation datasets. The classification pipeline can be seen as a complex classifier consisting of many components and processing steps.

A naïve variant of cross-validation for the classification pipeline would only consider the generalization of the classifier. First, the feature selection, feature preprocessing and the feature transform would be trained and applied

using the whole dataset T_{Train}. Then the highly processed dataset is separated into training and validation datasets for the classifier. This naïve variant is a suboptimal way to estimate the generalization of the whole pipeline as especially the feature transform element has a huge impact on the data representation. A highly non-linear problem might become linearly separable after the transform – the "intelligence" can move from the classifier to the feature transform. But on the other hand, if complex non-linear models are learned during manifold learning, there is always the risk of overfitting. Furthermore, a fraction of commonly used feature transforms is supervised, e.g., LDA or LMNN (see appendix C), and so they make use of the true class labels. It can be problematic when such feature transforms are learned with the full dataset T_{Train}: Consider a supervised feature transform that projects all vectors of class ω_1 to the value 1, all vectors of class ω_2 to 2 and so on. When all vectors in T_{Train} are transformed in such a way, the representation would be "perfect" in the sense that the classes would be trivially separable. However, there is no guarantee that new instance vectors will be transformed in such a perfect way since the true labels are not available. Therefore, the generalization of the feature transform has to be estimated as well.

Additionally, the feature preprocessing also contains model parameters that are learned from the feature data. For instance, even the simple feature rescaling method to a value range of $[0, 1] \subset \mathbb{R}$ is very sensitive to noise and outliers. A single large value in the feature data directly affects the minimum and maximum values leading to suboptimal scaling factors. Consider the following data of a feature dimension, e.g., $x_1^{(i)}$ for several instances with $1 \leq i \leq 6$

$$[0, \quad 1, \quad 2, \quad 3, \quad 4, \quad 10^6]\,. \tag{5.1}$$

The minimum value is 0 and the maximum value is 10^6 leading to rescaled features $x_{rescaled,1}^{(i)}$ of

$$[0, \quad 10^{-6}, \quad 2 \cdot 10^{-6}, \quad 3 \cdot 10^{-6}, \quad 4 \cdot 10^{-6}, \quad 1] \tag{5.2}$$

having most values very close to zero which might lead to numerical instabilities. Therefore, it is also reasonable to check the generalization of the preprocessing as well.

5.1.2 Holistic Cross-Validation

The previous section describes the necessity to incorporate all components of the pipeline, especially the feature transform, into the cross-validation process. This is achieved with an extended, holistic variant of cross-validation as described in algorithm 1. The algorithm needs a training dataset T_{Train} and a pipeline configuration θ. It returns an estimation of the generalization in form of the average overall accuracy, which can be directly used as the

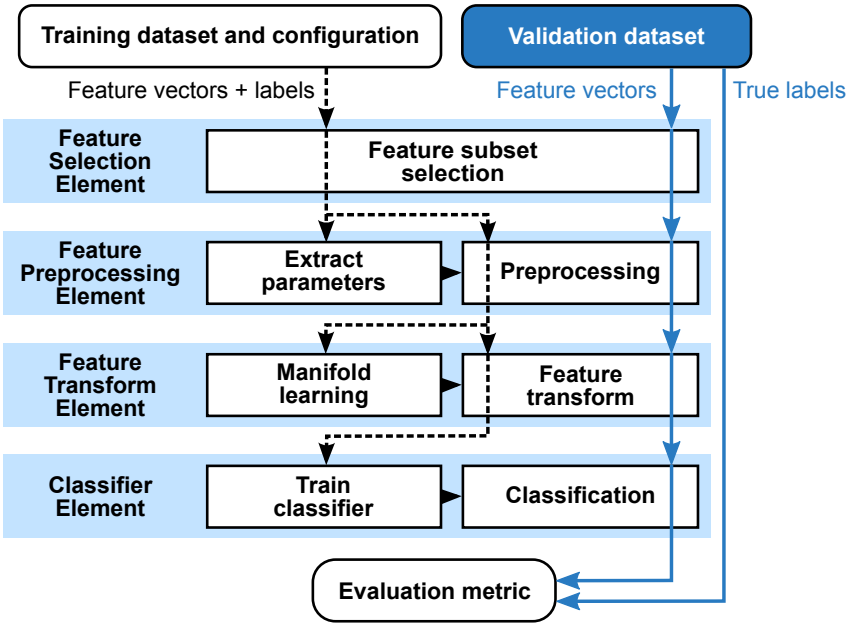


Figure 5.1: Processing schema of the proposed holistic cross-validation to estimate the generalization of all components of the classification pipeline. Note that training and validation dataset are always processed separately and no model parameters of any algorithm are estimated using the validation dataset.

optimization objective function (see section 5.2 and 5.5.2). The principle is similar to a standard cross-validation approach in which the dataset T_{Train} is also divided into N_{CV} disjoint parts resulting in $1 \leq j \leq N_{CV}$ training and validation dataset tuples $\{(T_{CV,Train,j}, T_{CV,Valid,j})\}$. Then up to N_{CV} evaluation rounds are performed while in each round a classification pipeline is trained with the training dataset $T_{CV,Train,j}$ and the configuration θ. It is important to note that all model parameters of all pipeline elements are trained by the training dataset only. The validation dataset $T_{CV,Valid,j}$ is afterwards processed separately through the trained pipeline and the predicted labels are used to calculate the average overall accuracy. Figure 5.1 depicts the processing stages of the training and validation datasets. Finally, the average overall accuracy of all rounds is returned. Note that the *earlyDiscarding* function can prematurely stop the cross-validation process (see section 5.1.3).

A rule of thumb in machine learning is the use of $N_{CV} = 10$ cross-validation rounds while usually only a classifier model is trained. However, as multiple algorithms need to be trained and processed in the classification pipeline, $N_{CV} = 5$ is used to save computation time.

Algorithm 1: Pseudocode of *holistic cross-validation*, denoted as function $holisticCV(\theta, T_{Train})$.

Data: training dataset T_{Train} and a pipeline configuration θ
Result: generalization estimation in form of the average overall accuracy $q_{OAcc,mean}$

$j := 1$
$continue :=$ true
$accuracyValues :=$ Array[] // dynamic array
while *continue* **do**
    // get training and validation datasets
    $T_{CV,Train,j} := getTrainingDataset(T_{Train}, j)$
    $T_{CV,Valid,j} := getValidationDataset(T_{Train}, j)$
    $\Theta := f_{TrainPipeline}(\theta, T_{CV,Train,j})$ // train pipeline
    // reset arrays of class label statistics
    $labelsPredicted :=$ Array$[N_{T_{CV,Valid,j}}]$
    $labelsTrue :=$ Array$[N_{T_{CV,Valid,j}}]$
    // classify all instances in validation dataset
    for $1 \leq l \leq N_{T_{CV,Valid,j}}$ **do**
        // get prediction of pipeline on lth feature vector
        $labelsPredicted[l] := f_{ProcessPipeline}(\Theta, \mathbf{x}^{(l)}_{CV,Valid,j})$
        $labelsTrue[l] := y^{(l)}_{CV,Valid,j}$ // true validation label
    // calculate accuracy statistics
    $q_{OAcc,j} := getOverallAccuracy(labelsTrue, labelsPredicted)$
    $accuracyValues[j] := q_{OAcc,j}$ // append current accuracy
    $q_{OAcc,mean,j} :=$ mean$(accuracyValues)$ // mean so far
    $j := j + 1$
    // termination criteria
    $continue := j \leq N_{CV} \ \wedge \ \neg earlyDiscarding$
// return accuracy values
$q_{OAcc,mean} := q_{OAcc,mean,j}$

5.1.3 Processing Speedup with Early Discarding

There is a large potential to save computation time as a simple observation reveals: It can be expected that a comparatively great number of configurations will perform relatively poorly or at least inferiorly compared to the current best solution during an optimization phase. This observation can be used to define early discarding criteria that will stop the cross-validation process for bad configurations as early as possible in order to save computation time. The idea is related to so-called *racing algorithms*, e.g., Hoeffding Races [Maron and Moore, 1994], which have been designed for efficient model selection. Racing algorithms do not necessarily evaluate all instances from the validation dataset. Instead, each instance is evaluated consecutively and the performance metric, e.g., the average overall accuracy, is calculated after each instance. The evaluation is stopped prematurely when the estimated performance metric is significantly lower compared to the best known models. The stop criterion is usually determined using statistical tests (see appendix F). The more "aggressive" such a racing algorithm is, the larger the speedup potential is. On the other hand, the risk of discarding good solutions increases.

It would be possible to use a racing algorithm for the evaluation of instances of the validation dataset during the cross-validation process. However, the training mode of the proposed classification pipeline has a much larger effect on the overall speed than the classification mode. During the training a feature preprocessing model, a feature transform and a classifier have to be trained. Especially the training process of many of the complex non-linear feature transform algorithms (see appendix C) is computationally expensive. Consequently, the speedup effect of a classical racing algorithm would be marginal.

Therefore, an *early discarding* system is not used for the validation instances, but during the cross-validation rounds themselves. This can be achieved with two early discarding conditions that are formulated as follows:

1. At first, all configurations leading to classifiers that perform worse than guessing[1] are not promising. The threshold of guessing depends on the number of classes and the first condition can be formulated as

$$discard_{Guess} = \begin{cases} 1: & q_{OAcc,j} < \frac{1}{N_{Classes}} \\ 0: & else \end{cases} \quad \text{for} \quad \exists j \in \{1, 2, \ldots, N_{CV}\}\,. \tag{5.3}$$

   Note that this condition is true even when only a single cross-validation round is affected while the following rounds could have improved the mean accuracy value. It is assumed that classifiers that perform badly at least once will not be very useful in practical applications.

[1] A classifier that randomly guesses class labels has a chance of $1/N_{Classes}$ to predict the correct label when the classes are equally probable. A real classifier should obviously perform better.

2. During the optimization process, multiple configurations are evaluated and the accuracy statistics of the previously best solution is stored as $q_{OAcc,mean,best}$. A quite aggressive discarding criterion can be defined so that the currently evaluated solution needs to be better or equal than the last best solution. Otherwise, it is assumed that the current solution is inferior and no more time should be wasted on further evaluations of it. During the N_{CV} cross-validation rounds, the current mean accuracy $q_{OAcc,mean,j}$ estimation is calculated. The discarding criterion is simply formulated as

$$discard_{AccuracyMean} = \begin{cases} 1: & q_{OAcc,mean,j} < q_{OAcc,mean,best} \\ 0: & else \end{cases} \tag{5.4}$$

for $\exists j \in \{1, 2, \ldots, N_{CV}\}$. Consider the following example in which $q_{OAcc,mean,best} = 0.85$ and a new configuration that would achieve these five accuracy values during the cross-validation: $q_{OAcc,1} = 0.84$, $q_{OAcc,2} = 0.9$, $q_{OAcc,3} = 0.9$, $q_{OAcc,4} = 0.9$ and $q_{OAcc,5} = 0.9$. The proposed criterion discards the continuation of the evaluation process after the first cross-validation round because the first accuracy value (0.84) is below the last best accuracy value (0.85). This leads to a performance underestimation because the true average overall accuracy for this configuration would be 0.888 – which is even better than the last best accuracy.

Finally, the two conditions are evaluated in the *earlyDiscarding* function in algorithm 1 which returns true if any of the two conditions is true

$$earlyDiscarding = discard_{Guess} \vee discard_{AccuracyMean}\,. \tag{5.5}$$

If this is the case, the cross-validation is stopped after this round and the average overall accuracy that has been estimated so far is returned as an estimation for the generalization. Note that configurations for which at least one of the two quite aggressive criteria has been true are not removed but that their generalization performance will most likely be underestimated.

However, it is assumed that a sufficiently large number of configurations will be evaluated during the optimization process and so it is likely that a similar configuration will finally replace the last best one. The randomization of the cross-validation dataset division for each configuration supports this assumption. Furthermore, the skipping of cross-validation rounds does not lead to an overoptimistic estimation of the generalization for practically useful configurations. In case the average cross-validation accuracy of a configuration is better or equal than the currently best one, the cross-validation process is not stopped prematurely. Therefore, it is ensured that the generalization estimation of the overall best configuration is not biased by the early discarding criteria.

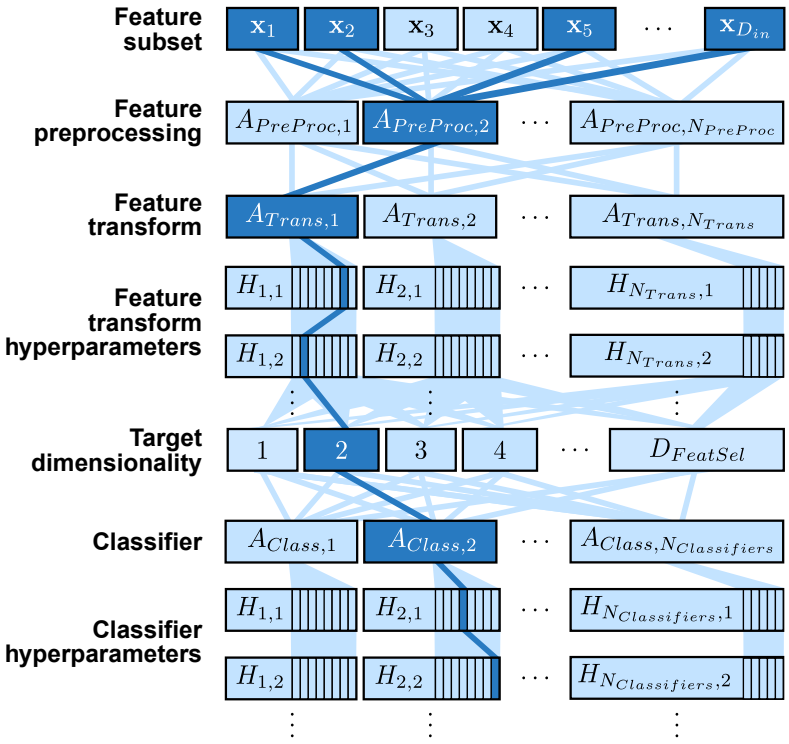


Figure 5.2: Visualization of the combinatorial and hierarchical configuration adaptation problem with one selected configuration (dark items) out of all possible combinations $S_\theta(T_{Train})$ (bright items).

5.2 The Configuration Adaptation Problem

The configuration adaptation problem is stated as

$$\theta^* = \arg\max_{\theta} \; g_{Obj,Gen}(\theta, T_{Train}) \quad \text{for} \quad \forall \theta \in S_\theta(T_{Train}) \tag{5.6}$$

to find the best configuration θ^* out of all possible configurations $S_\theta(T_{Train})$ with respect to an objective function $g_{Obj,Gen}$. The objective function needs to estimate the generalization of the pipeline configurations and therefore, the holistic cross-validation, proposed in the previous section, is used as objective function, i.e.

$$g_{Obj,Gen}(\theta, T_{Train}) = holisticCV(\theta, T_{Train})\,. \tag{5.7}$$

5.2.1 Structural Analysis

Figure 5.2 depicts the full combinatorial and hierarchical structure of the configuration adaptation problem and figure 5.3 shows the problem formulation as a high-dimensional optimization problem in which each degree of freedom

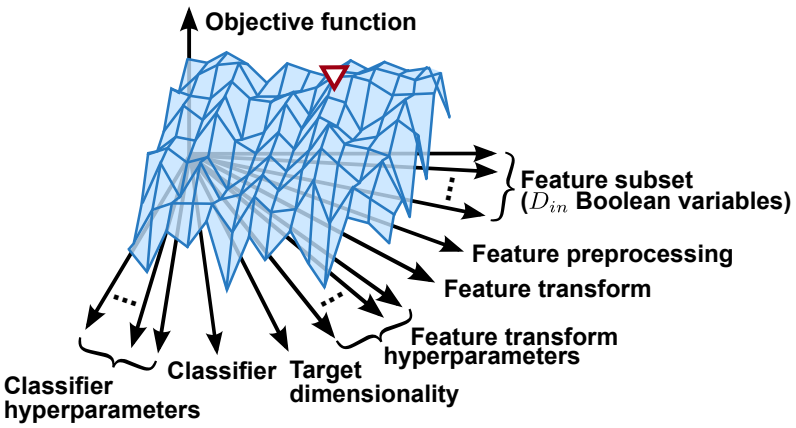


Figure 5.3: Visualization of the configuration adaptation problem as an optimization problem with a high-dimensional search space. Note that the objective function surface is only indicated as two-dimensional. A hypothetical optimal solution is indicated with a triangle.

in θ is an optimization variable. This task is particularly challenging due to several reasons:

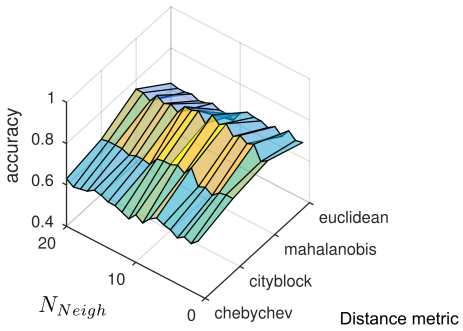
- The greatest challenge is the extremely large number of possible configurations (see section 5.2.2) due to the feature selection problem, that introduces D_{in} Boolean variables, one for each feature to be selected or not. Additionally, the algorithm portfolios in the feature preprocessing, feature transform and classifier element contribute to the combinatorial explosion as well.
- The problem is hierarchical because of the hyperparameters that are dependent on the selection of a specific algorithm. Hyperparameters are only meaningful when the corresponding algorithm is selected.
- The configuration contains many different variable types such as binary feature selection switches, categorical and numerical variables.
- The problem is very ill-posed and the objective function can only estimate the expected generalization of a pipeline configuration.
- The interplay between the algorithms is complex, small changes (like adding or removing a relevant feature or changing the preprocessing algorithm) can cause tremendous changes in the feature distribution for the classifier – and lastly degrade or improve the performance.

All components of a pipeline configuration θ are important. However, the impact on the classification performance with respect to small changes is expected to be different depending on the component:

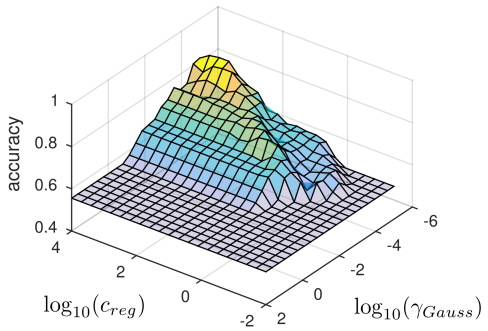
- *Feature subset* $S_{FeatSubSet}$: The adding or removal of a single, very relevant feature dimension can have a tremendous impact on the performance while the modification of a few less relevant features is expected to have few impact.
- *Feature preprocessing* $A_{PreProc}$: Datasets with very heterogeneous feature sources may lead to feature value ranges that differ in orders of magnitudes. In this case the change of the preprocessing method has a potentially huge impact.
- *Feature transform* A_{Trans}: The feature transform usually changes the representation significantly and thus its choice has a very strong impact on the performance. One feature transform can be fairly useful for classification while another one may even lead to a performance worse than guessing – for the same learning task.
- *Feature transform hyperparameters* $S_{\mathbb{H},A_{Trans}}$: These hyperparameters may have a large effect on the resulting transform, however, it depends on the type:
  - *Categorical variables* usually act as discrete "switches" to change modes within the algorithms, which can have a large impact on the performance.
  - *Numerical variables* also have an impact, of course, but small changes usually do not lead to huge differences.
- *Feature transform target dimensionality* D_{Trans}: Small changes of the target dimensionality are expected to influence the performance only scarcely. Some additional but irrelevant dimensions usually do not cause negative effects related to the curse of dimensionality. However, if the dimensionality becomes too low, important discriminative information might get lost.
- *Classifier* A_{Class}: The choice of the classifier has a huge impact according to the no-free-lunch theorem.
- *Classifier hyperparameters* $S_{\mathbb{H},A_{Class}}$: The impact is similar to the hyperparameters of the feature transform, which depends on the hyperparameter's type (see above).

An example of the different impact "strength" of specific components of the pipeline configuration on the classification performance is depicted in figure 5.4. The *Statlog (Heart)* dataset from the UCI[2] machine learning repository [Bache and Lichman, 2013] is used here as a relatively simple example. The dataset originates from a medical application in which symptoms and measurements of patients need to be correlated to the absence or presence

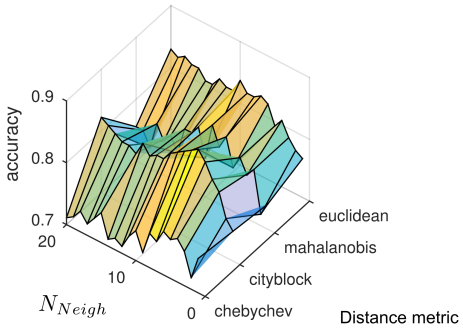
[2]University of California, Irvine.



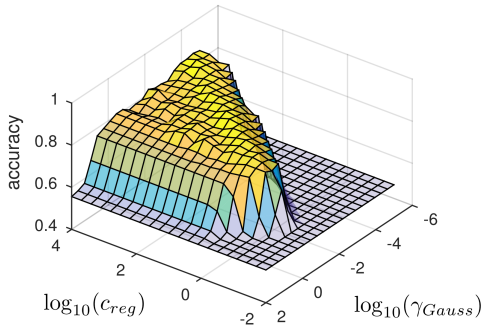
(a) PP: none, FT: none, Cl.: **kNN**



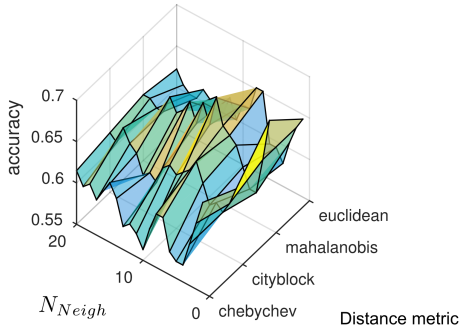
(b) PP: none, FT: none, Cl.: **SVM**



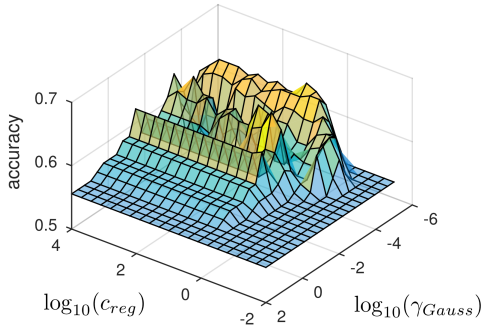
(c) PP: **rescaling**, FT: none, Cl.: **kNN**



(d) PP: **rescaling**, FT: none, Cl.: **SVM**



(e) PP: none, FT: **PCA**, Cl.: **kNN**



(f) PP: none, FT: **PCA**, Cl.: **SVM**

Figure 5.4: Comparison of cross-validation accuracy "landscapes" depending on different hyperparameter values and configurations (PP = Preprocessing, FT = feature transform, Cl. = classifier). The *Statlog (Heart)* dataset from the UCI machine learning repository [Bache and Lichman, 2013] is used.

of a serious heart disease. The configurations are evaluated with the average overall accuracy from the holistic cross-validation method (see section 5.1) using a grid search of the algorithms and hyperparameters.

It can be observed that categorical variables generally cause very discrete and abrupt changes in the accuracy distribution. This is especially the case for the algorithm selections, such as the choice of the preprocessing algorithm (none vs. rescaling), the feature transform algorithm (none vs. PCA) and the classifier (kNN vs. SVM with a Gaussian kernel). However, the distance metric of the kNN-classifier – a categorical hyperparameter – also shows a very discrete impact on the accuracy.

The impact of numerical hyperparameters, especially of the SVM, on the accuracy is rather smooth in most cases but may also cause steep peaks and valleys within the accuracy "landscapes". Note that the full configuration adaptation problem cannot be visualized easily, e.g., the impact of different feature subsets is not shown here, instead the full feature set is used.

In summary, even this simple example shows that the distribution of the objective function based on cross-validation is complex with many discontinuities. However, several partially smooth areas and plateaus around good solutions exist having a similar range of accuracy values across different configurations (around the value 0.8 in this case). This observation shows that several good configurations exist and that many machine learning algorithms are not extremely sensitive to changes regarding the hyperparameters or the feature representation. On the other hand, there is the challenge of numerous local optima of the objective function that could prevent finding the best, or at least near-optimal configurations.

5.2.2 Complexity Analysis

The figures 5.2 and 5.3 already give an impression of the combinatorial explosion of the configuration adaptation problem. This section analyzes the theoretical complexity of the problem. In the following, the number of possible combinations $|S_\theta(T_{Train})|$ depending on the training dataset T_{Train} is estimated. The complexity is derived starting with the algorithm and hyperparameter selection problem and ending with the feature selection problem.

The precise problem complexity cannot be described in a reasonable way as, e.g., real-valued hyperparameters can have infinitely many different values to choose from. Therefore, the hyperparameter selection problem is simplified in the following way to obtain a lower estimate of the complexity. A coarse grid sampling density is considered using $N_{GridSamples} = 3$ sample points for all hyperparameters regardless of their type. Note that this is a rather coarse grid – the grid sampling in figure 5.4, e.g., uses 20 samples for each numerical hyperparameter.

Two pipeline elements in the classification pipeline use hyperparameters, namely the feature transform element E_{Trans} and the classifier element $E_{Classifier}$. The algorithms in the feature preprocessing element $E_{PreProc}$ do not have any hyperparameters and are thus not considered. The number of

feature transform variants (algorithm and hyperparameter combinations) assuming grid sampling can be formulated as the sum of the product spaces

$$N_{HypCombis,Trans} = \sum_{j=1}^{N_{Trans}} N_{GridSamples}^{N_{Hyp,Trans,j}} \tag{5.8}$$

in which N_{Trans} denotes the number of feature transforms in the portfolio and $N_{Hyp,Trans,j}$ the number of hyperparameters of the jth algorithm. If a method does not have any hyperparameter, then the sum is only incremented by $N_{GridSamples}^{0} = 3^0 = 1$.

The number of classifier variants (classifiers and hyperparameter combinations) can be treated in the same way using

$$N_{HypCombis,Class} = \sum_{l=1}^{N_{Classifiers}} N_{GridSamples}^{N_{Hyp,Class,l}} \tag{5.9}$$

with $N_{Classifiers}$ as the number of classifiers in the portfolio and $N_{Hyp,Class,l}$ as the number of hyperparameters of the lth classifier.

Using these equations the first approximation of the number of combinations can be expressed as

$$|S_\theta(T_{Train})| \approx \underbrace{\left(2^{D_{in}} - 1\right)}_{\substack{\text{Feature} \\ \text{selection}}} \cdot \underbrace{N_{PreProc}}_{\text{Preprocessing}} \cdot \underbrace{N_{HypCombis,Trans} \cdot D_{in}}_{\substack{\text{Feature transform,} \\ \text{target dimensionality}}} \cdot \underbrace{N_{HypCombis,Class}}_{\text{Classifier}} \tag{5.10}$$

in which the product space of the feature subset selection problem, the number of preprocessing algorithms, the number of feature transform variants with the target dimensionality in the range of $1 \leq D_{Trans} \leq D_{in}$ (hence the factor D_{in}) and finally the number of classifier variants is considered.

However, the possible values of the target dimensionality D_{Trans} for the feature transform actually depend on the number of selected features $D_{FeatSel}$ in the feature selection element $E_{FeatSel}$ as only a dimension *reduction* is allowed for the feature transform. Considering this constraint of $1 \leq D_{Trans} \leq D_{FeatSel} \leq D_{in}$, a second, more precise approximation for the number of combinations can be defined as

$$|S_\theta(T_{Train})| \approx \underbrace{\left(\sum_{k=1}^{D_{in}} k \cdot \binom{D_{in}}{k}\right)}_{\substack{\text{Feature selection,} \\ \text{target dimensionality}}} \cdot \underbrace{N_{PreProc}}_{\text{Preprocessing}} \cdot \underbrace{N_{HypCombis,Trans}}_{\substack{\text{Feature} \\ \text{transform}}} \cdot \underbrace{N_{HypCombis,Class}}_{\text{Classifier}} \tag{5.11}$$

combining the number of feature combinations with the constraints of the target dimensionality. The term $\binom{D_{in}}{k}$, "D_{in} choose k", describes how many

different combinations of k features can be selected from D_{in} total features – this number is multiplied with k to include the number of possible values for the target dimensionality $1 \leq D_{Trans} \leq k$. The sum over all $1 \leq k \leq D_{in}$ values takes the complete feature selection problem into account. However, the overall complexity compared to the first approximation does only change by a factor of $\frac{1}{2}$ as an algebraic transformation reveals that

$$\sum_{k=1}^{D_{in}} k \cdot \binom{D_{in}}{k} = D_{in} 2^{D_{in}-1} = \frac{1}{2} D_{in} 2^{D_{in}} \approx \frac{1}{2} D_{in} \left(2^{D_{in}} - 1\right). \tag{5.12}$$

The overall complexity can be approximated as

$$|S_\theta(T_{Train})| = \mathcal{O}\left(\phi \cdot D_{in} 2^{D_{in}}\right) = \mathcal{O}\left(\phi \cdot 2^{D_{in}+\log_2(D_{in})}\right) \tag{5.13}$$

in which ϕ is a constant depending on the algorithm portfolios and their hyperparameters. Within this work $N_{PreProc} = 5$ preprocessing algorithms, $N_{Trans} = 31$ feature transforms and $N_{Classifiers} = 8$ classifiers are used. Using a sample grid with $N_{GridSamples} = 3$ and roughly assuming that every feature transform and every classifier has one hyperparameter on average[3], the factor ϕ can be approximated as $\phi \approx \frac{1}{2} \cdot 5 \cdot (31 \cdot 3^1) \cdot (8 \cdot 3^1) = 5{,}580$. The factor $\frac{1}{2}$ originates from equation 5.12.

With these assumptions the structure of the optimization variables can be estimated as

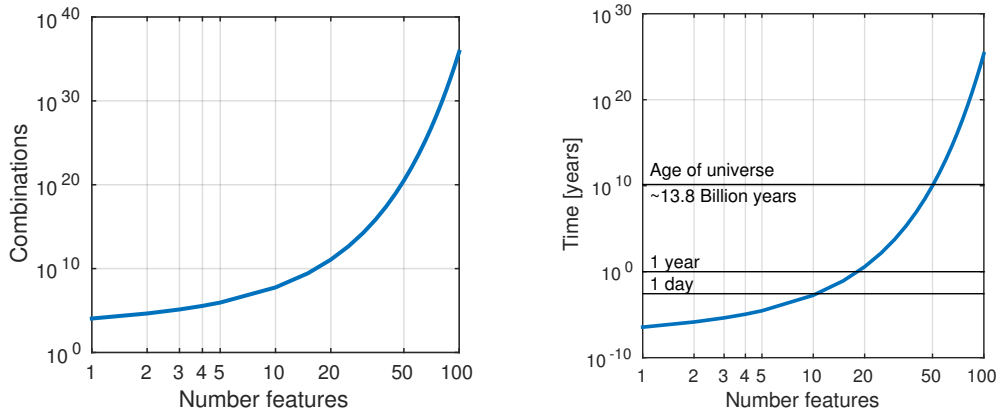
- D_{in} variables for the feature selection,
- 1 variable for the feature preprocessing method,
- 1 variable for the feature transform method,
- $\mathcal{O}(N_{Trans})$ variables for the feature transform hyperparameters,
- 1 variable for the target dimensionality,
- 1 variable for the classifier and
- $\mathcal{O}(N_{Classifiers})$ variables for the classifier hyperparameters.

This leads to a total number of optimization variables of

$$N_{Opt} = \mathcal{O}(D_{in} + N_{Trans} + N_{Classifiers} + 4) \tag{5.14}$$

and therefore, $N_{Opt} = \mathcal{O}(D_{in} + 31 + 8 + 4) = \mathcal{O}(D_{in} + 43)$ when the algorithm portfolios of this work are considered. Figure 5.5 shows the combinatorial explosion depending on the feature space dimensionality. The hypothetical optimization runtime is estimated for a brute force grid search approach at 0.001 seconds per evaluation of a configuration – which can be considered as very fast. Problems with $D_{in} \approx 20$ features would already require more than one year of processing time. It is almost needless to say that this naïve approach is infeasible for real-world problems as, e.g., the standard LBP texture descriptor in image processing already has 256 dimensions.

[3] In total there are 40 hyperparameters for 31 feature transforms.



(a) Number of combinations

(b) Hypothetical optimization runtime

Figure 5.5: Visualization of the combinatorial explosion of the configuration adaptation problem depending on the number of features D_{in}.

5.3 Suitable Optimization Algorithms

In order to find suitable optimization algorithms for the configuration adaptation problem, the following requirements have to be at least partially fulfilled:

- Finding the *global* optima of the objective function is desirable, but likely impossible when looking at the number of possible combinations in figure 5.5. However, near-optimal configurations are expected to be sufficient.
- The optimization algorithm has to be able to handle a high number of optimization variables. More than 1,000 variables for image-based classification tasks are realistic.
- Different variable types need to be optimized simultaneously: Boolean, categorical, real-valued and integer variables.
- The hierarchical structure of the hyperparameters depending on the algorithm selection needs to be handled.
- The challenge of a complex objective function with discontinuities and numerous local optima has to be handled. A general smoothness property of the objective function cannot be assumed.
- A relatively fast optimization runtime is desirable, however, several hours would be tolerable as it may run during the night. The optimiza-

tion process is only needed once in the design phase of the classification system[4].

- The optimization algorithm itself should have as few metaparameters as possible so that only few expert knowledge is required to use it. The optimization process should work completely automatically in the best case.

Section 3.1 provides an overview of popular optimization heuristics for complex optimization problems. In the following, the suitability of these approaches for the configuration adaptation problem is discussed.

5.3.1 Simple Optimization Methods

A suitable search strategy for selecting and evaluating configurations needs to be employed when a wrapper approach is used. As mentioned in the previous section, a naïve *exhaustive grid search*, also called *brute force strategy*, is clearly infeasible.

The largest contribution to the problem complexity arises from the feature selection problem. One idea is to use heuristic feature selection algorithms such as sequential feature selection (see section 3.2.2), which are polynomial time algorithms. The other parts of the search space can be treated with grid search. This hybrid approach was implemented in a previous version of the proposed framework [Bürger et al., 2014] with a much smaller set of algorithms, namely only the PCA in the set of feature transforms as well as six classifiers. The limitations of the remaining problem complexity were obvious and a further extension of more optimization variables could be considered as not promising. Obviously, more sophisticated optimization methods are required to adequately tackle the configuration adaptation problem.

5.3.2 Metalearning

Metalearning approaches make use of experiences from previous learning tasks and are used to optimize classifiers and hyperparameters. One of the greatest challenges with such an approach is that a large database of solved learning tasks is required to cover a considerable amount of applications. It can be expected that this database will need to grow when the system adaptability increases – as it is clearly the case for the classification pipeline of this work. Additionally, the feature selection problem is hardly considered in the field of metalearning. Consequently, the idea of metalearning is potentially tempting but obviously not directly suitable.

[4]Once a configuration is found and the classification pipeline is initialized, the processing to classify an instance runs at the speed of the chosen algorithms – usually a few milliseconds per instance. The classification times are also discussed in the evaluation chapter (see section 7.5.3).

5.3.3 Trajectory-based Approaches

Trajectory-based search approaches are potentially suitable for the configuration adaptation problem. Starting with an initial, e.g., random configuration, the next most promising configuration is evaluated and refined. Some approaches rely on gradient or derivative information of the objective function to find the direction of the next promising optimum, e.g., the backpropagation learning algorithm of artificial neural networks. However, these methods are unsuitable here as the objective function is highly discontinuous (see figure 5.4). The gradient information would be extremely noisy, especially due to the categorical optimization variables.

Model-based optimization and especially Sequential Model-Based Optimization (SMBO) are suitable to solve this problem. However, some limitations make this approach less attractive for the configuration adaptation problem. Due to the high complexity of the objective function it can be doubted that smooth Gaussian priors are adequate to model this function. Random forests might be better suitable, however, they would need a large number of training samples – configuration evaluations in this case – to describe the distribution of the objective function in a reasonable way. But even then, due to the complex interplay of the pipeline components, it is hard to predict the performance of unknown algorithm combinations. Furthermore, due to the sequential character of this method, it is not trivial to make use of parallelization on today's multi-processor computer systems.

5.3.4 Population-based Approaches

Population-based search algorithms are especially suitable for complex optimization problems as they are successfully used for hyperparameter optimization (see section 3.2.4) and algorithm configuration problems (see section 3.2.5). A great advantage of these algorithms is the parallel analysis of multiple promising areas of the search space. The random character of these heuristics potentially helps to escape from local optima of the objective function. A negative aspect is the relatively large number of evaluations of the objective function that might be needed to find good solutions. However, today's cheaply available multi-processor systems allow a parallel processing of many solutions at the same time. Four variants of these algorithms exist whose suitability regarding the configuration adaptation problem differ:

- Genetic Algorithms only use binary variables for the representations of solutions. This requires a partly cumbersome and less efficient coding schema of, e.g., numerical floating point variables.
- Genetic Programming allows the coding of whole programs, which is likely too flexible for the configuration adaptation problem as the general structure of the pipeline is fixed.

- Evolutionary Programming does not specify the representation of the solutions. So any kind of variables could be involved and it is potentially suitable for the configuration adaptation problem. However, a disadvantage is that Evolutionary Programming does not involve a recombination operator, which is an important aspect to explore new areas of the search space.
- Evolution Strategies are designed especially for general parameter optimization, but the basic variant is only defined for continuous, real-valued variables. However, the concept can potentially be extended for other variable types. Furthermore, all crucial evolutionary operators are involved which is certainly an advantage for difficult optimization problems.

Therefore, an optimization algorithm based on Evolution Strategies is expected to be promising for the configuration adaptation problem. The necessary extensions are presented in the following section.

5.4 Extended Evolution Strategies

Evolution Strategies are a variant of Evolutionary Algorithms which have been developed in the 1960s by Rechenberg and Schwefel. The originally intended purpose of these methods was the parameter optimization of physical systems such as the minimization of the air resistance of objects [Rechenberg, 1965]. The methods are elaborated in dissertations [Rechenberg, 1973] and [Schwefel, 1977]. For a comprehensive and more up-to-date introduction, the reader is kindly referred to textbooks like [Bäck, 1996], [Beyer and Schwefel, 2002] or [Kramer, 2008]. The basic terms of Evolutionary Algorithms have already been listed in section 3.1.1. The following subsections describe extensions for the standard Evolution Strategies which are mostly targeted to be useful for the configuration adaptation problem. However, there is no general limitation of the proposed *extended Evolution Strategies* for a specific optimization problem.

5.4.1 Variable Representations

A central aspect for all Evolutionary Algorithms is the genetic representation of a solution for the optimization problem. The originally proposed Evolution Strategies only support vectors of continuous numbers in $\mathbb{R}^{N_{Opt}}$. This is not sufficient to describe a solution of the configuration adaptation problem in an adequate way. However, the concept of Evolution Strategies is potentially open to extensions, which was already shown in [Müller, 2012]. Inspired by the existing extensions, five different types of variables and corresponding

properties are described in the following. The properties contain value constraint information as well as mutation parameters which are required for the evolutionary operators (see section 5.4.2).

- *Continuous, linearly scaled, real-valued variables* are the originally used variable type in Evolution Strategies. This variable type is defined as

$$V_{\mathbb{R}} = (v_{\mathbb{R}},\ [v_{\mathbb{R}}^{min},\ v_{\mathbb{R}}^{max},\ \sigma_{Mut,\mathbb{R}}]) \tag{5.15}$$

  in which $v_{\mathbb{R}} \in \mathbb{R}$ is the actual value that lies within the value constraints $v_{\mathbb{R}}^{min} \leq v_{\mathbb{R}} \leq v_{\mathbb{R}}^{max}$ with $v_{\mathbb{R}}^{min} \in \mathbb{R}$ and $v_{\mathbb{R}}^{max} \in \mathbb{R}$. The value of $\sigma_{Mut,\mathbb{R}} \in \mathbb{R}$ is the mutation parameter for this variable. Note that scalar numeric variables are sufficient for the configuration adaptation problem. However, if a vector is needed for other optimization problems, it could be split into multiple scalars.

- *Exponentially scaled, real-valued variables* are similar to linearly scaled, continuous variables. A popular example is the regularization hyperparameter c_{reg} of the SVM, which has a typical value range of $[10^{-2}, 10^{4}] \subset \mathbb{R}^{+}$. It is advantageous to handle these variables differently in the evolutionary optimization process because, e.g., the average of two values needs to be computed in a reasonable way[5]. Therefore, the following definition is used

$$V_{exp\mathbb{R}} = (v_{exp\mathbb{R}},\ [v_{exp\mathbb{R}}^{min},\ v_{exp\mathbb{R}}^{max},\ \sigma_{Mut,exp\mathbb{R}}]) \tag{5.16}$$

  in which $v_{exp\mathbb{R}} \in \mathbb{R}$ is the log_{10}-exponent of the actual hyperparameter value that can be obtained using $\tilde{v}_{exp\mathbb{R}} = 10^{v_{exp\mathbb{R}}}$. Note that the exponentially scaled hyperparameters usually have a value domain of $\tilde{v}_{exp\mathbb{R}} \in (0, \infty] \subseteq \mathbb{R}^{+}$ so that the logarithm values are always well defined. However, $v_{exp\mathbb{R}}$, the log_{10}-exponent, becomes negative for $\tilde{v}_{exp\mathbb{R}} < 1$. The variable constraints are defined as $v_{exp\mathbb{R}}^{min} \leq v_{exp\mathbb{R}} \leq v_{exp\mathbb{R}}^{max}$ with $v_{exp\mathbb{R}}^{min} \in \mathbb{R}$ and $v_{exp\mathbb{R}}^{max} \in \mathbb{R}$. Note that these constraints apply for the exponents and the actual hyperparameter values obey the following constraints $\tilde{v}_{exp\mathbb{R}}^{min} \leq \tilde{v}_{exp\mathbb{R}} \leq \tilde{v}_{exp\mathbb{R}}^{max}$ with $\tilde{v}_{exp\mathbb{R}}^{min}, \tilde{v}_{exp\mathbb{R}}^{max} \in \mathbb{R}^{+}$. The limits of the exponents can be calculated using $v_{exp\mathbb{R}}^{min} = \log_{10}(\tilde{v}_{exp\mathbb{R}}^{min})$ and $v_{exp\mathbb{R}}^{max} = \log_{10}(\tilde{v}_{exp\mathbb{R}}^{max})$.

  The mutation parameter is $\sigma_{Mut,exp\mathbb{R}} \in \mathbb{R}$. The advantage of this definition is that the exponentially scaled variables can be used in the same way as the linearly scaled ones in the evolutionary operators.

[5]The "normal" average of, e.g., 10^4 and 10^{-2} is $5{,}000.005 \approx 5 \cdot 10^3$, which is very close to first value (10^4). As large values dominate small values, a more meaningful average is obtained using the average of the exponents, i.e. $10^{(4-2)/2} = 10^1 = 10$.

- *Integer variables* often occur in hyperparameters, e.g., the number of neighbors N_{Neigh} in the kNN classifier. The definition is analogous to the continuous variables with

$$V_{\mathbb{Z}} = (v_{\mathbb{Z}},\ [v_{\mathbb{Z}}^{min},\ v_{\mathbb{Z}}^{max},\ \sigma_{Mut,\mathbb{Z}}]) \tag{5.17}$$

  in which $v_{\mathbb{Z}} \in \mathbb{Z}$ is the actual value inside of the constraints $v_{\mathbb{Z}}^{min} \leq v_{\mathbb{Z}} \leq v_{\mathbb{Z}}^{max}$ with $v_{\mathbb{Z}}^{min} \in \mathbb{Z}$ and $v_{\mathbb{Z}}^{max} \in \mathbb{Z}$. The mutation parameter is $\sigma_{Mut,\mathbb{Z}} \in \mathbb{R}$.

- *Boolean variables* are needed for the feature selection as each feature can be activated or deactivated. The variable type is defined as

$$V_{\mathbb{B}} = (v_{\mathbb{B}},\ [p_{Init,\mathbb{B}},\ p_{Mut,\mathbb{B}}]) \tag{5.18}$$

  in which $v_{\mathbb{B}} \in \{0, 1\}$ is the Boolean value. The value $p_{Init,\mathbb{B}} \in [0, 1] \subset \mathbb{R}$ determines the initial probability of setting this particular Boolean variable to the value *true*. Furthermore, the variable $p_{Mut,\mathbb{B}} \in [0, 1] \subset \mathbb{R}$ is the mutation parameter.

- *Categorical variables* are also important to select, e.g., an algorithm from a portfolio. This variable type is defined as

$$V_{\mathbb{S}} = (v_{\mathbb{S}}, [S_{\mathbb{S}},\ p_{Mut,\mathbb{S}}]) \tag{5.19}$$

  in which $v_{\mathbb{S}} \in S_{\mathbb{S}}$ is the selection of an item from the base set $S_{\mathbb{S}}$ that contains all possible values. The value $p_{Mut,\mathbb{S}} \in [0, 1] \subset \mathbb{R}$ is the corresponding mutation parameter.

5.4.2 Algorithm Overview

The genotype is the formal representation – or coding schema – of problem solutions in the (extended) Evolution Strategies. The variable types presented in section 5.4.1 are the building blocks of the genotype, which is defined as a sequence of N_{Genes} variables

$$\mathbb{G} = (V_{*,1}, V_{*,2}, \ldots, V_{*,N_{Genes}}) \tag{5.20}$$

in which $V_{*,j} \in \{V_{\mathbb{R}}, V_{exp\mathbb{R}}, V_{\mathbb{Z}}, V_{\mathbb{B}}, V_{\mathbb{S}}\}$. Note that the genotype itself does not contain a concrete solution. An individual $I \in \mathbb{G}$ has the same structure but contains a concrete solution with variable values that obey the variable constraints defined in the genotype. A population at the time index $t \in \mathbb{N}$ is a set of $\mu \in \mathbb{N}$ individuals

$$P_t = \{I_1, I_2, \ldots, I_\mu\}. \tag{5.21}$$

The general processing schema of Evolution Strategies is depicted in figure 5.6. Starting from an initial set of solution candidates (population), the

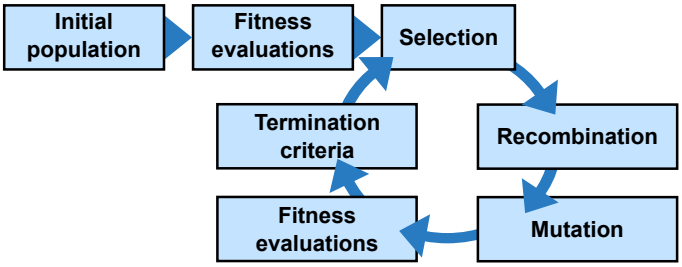


Figure 5.6: General processing loop of Evolution Strategies according to [Eiben and Smith, 2003].

fitness of each individual is evaluated using the objective function. Then the generation cycle starts in which, at first, the best individuals are selected for mating. The mating process creates a set of offspring individuals that contain a recombination of the parent genes. These genes are further mutated before the fitness of the new individuals is evaluated. Then, a set of termination criteria is checked to stop the evolution process after a certain amount of generations. If this is not the case, the cycle restarts with selecting the best individuals.

The most important metaparameters of Evolution Strategies are the following. The number of surviving individuals in the parent generation is denoted as μ. A small number for μ increases the selection pressure for good individuals, which can result in a faster solution improvement. However, a too high selection pressure turns the evolutionary process into a greedy algorithm and the risk of getting stuck in local optima increases.

The number of children that are generated in each generation is denoted as $\lambda \in \mathbb{N}$. A high number of children is generally desired as it increases the likelihood to improve the currently best solution. However, the computational cost increases as well. Each child is generated from $\rho \in \mathbb{N}$ parent individuals. The number of parents ρ is not – like in nature – limited to two parents. It controls the diversity of the new children – the more parents are involved, the greater the variety of the genetic information is and also the larger the newly explored search space is.

The maximum lifespan of individuals is defined as $\kappa \in \mathbb{N}$. A limited lifespan increases the diversity within the population when relatively fit individuals are already found by chance in early generations. These individuals would be chosen for mating disproportionately often and therefore, other solutions are hindered to evolve – the risk of getting stuck in local optima increases.

There are several important variants of Evolution Strategies that mainly differ regarding the selection operator. These can be described using a historically established short notation:

- $(\mu/\rho, \lambda)$ denotes the *comma selection schema* in which the selection operator only selects from the offspring while the parent generation dies ($\kappa = 1$).
- $(\mu/\rho+\lambda)$ denotes the *plus selection schema* in which the selection takes individuals from the parent generation and the offspring. The lifespan of individuals is unlimited ($\kappa = \infty$).
- $(\mu, \kappa, \lambda, \rho)$ is the most flexible selection schema which allows the adjustment of the maximal lifespan of individuals. It selects individuals from the parent and the offspring generation.

The proposed extended Evolution Strategies for the configuration adaptation problem are based on the most flexible $(\mu, \kappa, \lambda, \rho)$ variant. Algorithm 2 describes the processing schema and the listed steps are described in detail in the following subsections.

Algorithm 2: Pseudocode of extended Evolution Strategies.

Data: Evolutionary metaparameters $(\mu, \kappa, \lambda, \rho)$, genotype $\mathbb{G}$ and fitness/optimization function
Result: (Final) population P_t with corresponding fitness

```
1  t := 0
2  Initialize population P_t according to 𝔾
3  Assign fitness to all individuals in P_t
4  repeat
5      S_Offspring := {}
6      for 1 ≤ j ≤ λ do
7          Select a set S_Parents of ρ parents out of P_t
8          Generate individual I_j by recombination from S_Parents
9          Mutate individual I_j
10         Assign fitness to individual I_j
11         S_Offspring ← S_Offspring ∪ I_j
12     P_t ← P_t ∪ S_Offspring
13     Increase age of all individuals in P_t by one generation
14     Remove individuals with age > κ in P_t
15     Select the fittest μ individuals from P_t and remove the rest
16     t ← t + 1
17 until termination criteria
```

Population Initialization

First of all, the population P_0 has to be initialized. This is usually done by randomly generating μ_{init} individuals, if no other knowledge about the optimization problem at hand is available. However, if information is available,

it should be used in the initialization to obtain a better starting position for the search process, which is expected to result in a faster optimization and better fitness values. Therefore, the extended Evolution Strategies have two initialization modes: A completely random initialization mode, which is described in the following, and the possibility to use prior knowledge to improve the initial population, which is described in section 5.5.4.

Random Initialization Mode

The random initialization is a function

$$I = generateIndividual(\mathbb{G}) \tag{5.22}$$

that generates a new individual. A random variable value is determined for each item in the variable sequence defined by $\mathbb{G}$ according to its type:

- For continuous, linear variables, a uniformly distributed random number $v_{\mathbb{R}} \sim \mathcal{U}_{\mathbb{R}}(v_{\mathbb{R}}^{min}, v_{\mathbb{R}}^{max})$ is used.
- Exponentially scaled, continuous variables are handled in the same way as their linear counterparts with $v_{exp\mathbb{R}} \sim \mathcal{U}_{\mathbb{R}}(v_{exp\mathbb{R}}^{min}, v_{exp\mathbb{R}}^{max})$.
- Integer variables are also handled similarly, but with random integers $v_{\mathbb{Z}} \sim \mathcal{U}_{\mathbb{Z}}(v_{\mathbb{Z}}^{min}, v_{\mathbb{Z}}^{max})$.
- For Boolean variables a random binary value has to be set which is controlled by the corresponding initial probability $p_{Init,\mathbb{B}}$. This probability is set to $p_{Init,\mathbb{B}} = 0.5$ when no other prior knowledge is available. Let $rand \sim \mathcal{U}_{\mathbb{R}}(0,1)$ be a random number that determines the Boolean value with
$$v_{\mathbb{B}} = \begin{cases} 1: & rand \leq p_{Init,\mathbb{B}} \\ 0: & else \end{cases} . \tag{5.23}$$
- Categorical variables require the selection of a random item out of the base set $S_{\mathbb{S}}$ using
$$v_{\mathbb{S}} = randomItem(S_{\mathbb{S}}) . \tag{5.24}$$
The implementation of this function is realized by drawing a random integer $j_{rand} \sim \mathcal{U}_{\mathbb{Z}}(1, |S_{\mathbb{S}}|)$ that determines the index of the random item.

Fitness Assignment

All individuals that are created during the optimization need the assignment of a corresponding fitness value using the fitness evaluation function

$$fit(I) \mapsto \mathbb{R}^{+}. \tag{5.25}$$

The fitness function is usually directly connected to the desired optimization objective function. After the assignment of the fitness values for all individuals in the population P_t, a fitness vector

$$\mathbf{fit}(P_t) = [fit(I_1), fit(I_2), \ldots, fit(I_\mu)] \in \mathbb{R}^\mu \tag{5.26}$$

is available that is needed for further steps of the algorithm. Well performing individuals survive multiple generations and in order to save computation time the fitness function of these individuals is only evaluated once.

Selection Operator

The selection operator is crucial to select the best individuals out of a population. The fitter an individual is, the more likely will it be chosen for generating offspring. The proposed extension of Evolution Strategies makes use of the so-called *roulette wheel selection* schema [Goldberg, 1989], which is also known as *fitness proportionate selection*. The basic idea is similar to the roulette game and the sizes of the sections on the wheel are chosen according to the fitness values.

A fitness vector $\mathbf{fit}$ with μ values greater or equal to zero is given. It would be possible to use these *absolute* fitness values directly, but there are some undesired properties: If the fitness values are relatively close to each other, the likelihood to choose the worst individual is very similar compared to the fittest one. Therefore, a *relative* roulette wheel selection is used that transforms the fitness values linearly to a specific range in $fit_{low} < fit_{high}$ before with $fit_{low}, fit_{high} \in [0, 1] \subset \mathbb{R}$. The values $fit_{low} = 0.25$ and $fit_{high} = 1$ are chosen, so that the likelihood to choose the best individual is always four times higher than the chance to select the worst one. This increases the likelihood to select better solutions more often and thus leads to faster optimization runtimes. The minimum and maximum values of the input fitness vector are calculated and a scaling factor is obtained by

$$fit_{scale} = \frac{fit_{high} - fit_{low}}{\max(\mathbf{fit}) - \min(\mathbf{fit})}\,. \tag{5.27}$$

The transformed fitness vector is obtained by

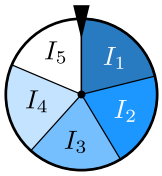
$$\begin{aligned}\mathbf{fit}_{rel} = [&fit_{low} + fit_{scale} \cdot (fit(I_1) - \min(\mathbf{fit})),\\ &fit_{low} + fit_{scale} \cdot (fit(I_2) - \min(\mathbf{fit})), \ldots,\\ &fit_{low} + fit_{scale} \cdot (fit(I_\mu) - \min(\mathbf{fit}))]\,.\end{aligned} \tag{5.28}$$

The sum of all relative fitness values

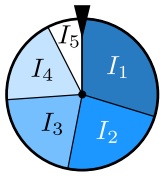
$$fit_{sum} = \sum_{j=1}^{\mu} \mathbf{fit}_{rel}(j) \tag{5.29}$$

	I_1	I_2	I_3	I_4	I_5
Fitness	0.85	0.82	0.81	0.8	0.75
p absolute	0.211	0.203	0.201	0.199	0.186
p relative	0.299	0.231	0.209	0.187	0.075

(a) Fitness and probability values



(b) Absolute



(c) Relative

Figure 5.7: Visualization of a roulette wheel selection example using the absolute and the relative variant. Table (a) lists the fitness values and the probabilities for each individual for both variants. The roulette wheels depicted in (b) and (c) illustrate these probabilities.

is used to norm the fitness values

$$\mathbf{fit}_{norm} = \frac{1}{fit_{sum}}[\mathbf{fit}_{rel}(1), \mathbf{fit}_{rel}(2), \ldots, \mathbf{fit}_{rel}(\mu)] \tag{5.30}$$

so that $\sum_{j=1}^{\mu} \mathbf{fit}_{norm}(j) = 1$. A cumulated sum vector

$$\mathbf{fit}_{cumulated} = \left[\sum_{j=1}^{1} \mathbf{fit}_{norm}(j), \sum_{j=1}^{2} \mathbf{fit}_{norm}(j), \ldots, \sum_{j=1}^{\mu} \mathbf{fit}_{norm}(j)\right] \tag{5.31}$$

is calculated to determine the probability borders of the roulette wheel sections on an interval between zero and one. To select an individual, a random number $rand \sim \mathcal{U}_{\mathbb{R}}(0,1)$ is drawn and the smallest index j_{select} with $\mathbf{fit}_{cumulated}(j_{select}) \geq rand$ is determined to return the individual $I_{j_{select}}$.

Figure 5.7 shows the effect of the absolute and relative variant of roulette wheel selection. In the example five individuals are considered and their fitness values can be found in figure 5.7 (a). The individual I_5 has a significantly lower fitness compared to the best one which is equal to ten percentage points less accuracy. In case of the absolute selection variant in figure 5.7 (b), the size of the section on the wheel for I_5 is almost similar to I_1. The relative variant in figure 5.7 (c) shows a much smaller section for I_5, making it much less likely – but not impossible – to choose this particular individual.

Recombination Operator

New individuals are created using the recombination and the mutation operator. First, ρ parent individuals are selected and their properties are recombined to form the basis for a new offspring individual. As already mentioned, the number of parents ρ is not limited to two. Furthermore, the parents of a single mating process do not necessarily need to be different individuals when one individual is selected multiple times.

The actual recombination process works in the following way. A set of ρ indices $S_{Parents,Indices} = \{j_1, j_2, \ldots, j_\rho\}$ with $1 \leq j_k \leq \mu$ and $1 \leq k \leq \rho$

is determined using the relative roulette wheel selection method. Then each variable in the genotype $\mathbb{G}$ is recombined according to its type:

- The average of the parent values is calculated for all numerical variables that is weighted by their relative fitness

$$v_{\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\},recom} = \frac{1}{\sum_{k=1}^{\rho} \mathbf{fit}_{rel}(j_k)} \sum_{k=1}^{\rho} \mathbf{fit}_{rel}(j_k) \cdot v_{\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\}}^{(j_k)} \tag{5.32}$$

  in which $v_{\{\cdot\}}^{(j_k)}$ denotes the specific variable value of the j_kth individual in P_t. After the recombination process the result is rounded for integer variables to obtain a valid integer.

- For Boolean and categorical variables a roulette wheel selection is performed to select a value out of the set $\{v_{\{\mathbb{B},\mathbb{S}\}}^{(j_1)}, v_{\{\mathbb{B},\mathbb{S}\}}^{(j_2)}, \ldots, v_{\{\mathbb{B},\mathbb{S}\}}^{(j_\rho)}\}$ according to the parents' relative fitness. The resulting value is denoted as $v_{\mathbb{B},recom}$ and $v_{\mathbb{S},recom}$, respectively. Note that this approach resembles the so-called uniform cross-over method [Syswerda, 1989, Bäck, 1996] in Genetic Algorithms when a bitstring of multiple Boolean variables is considered.

Each variable has a mutation parameter which is basically a continuous value, namely $\sigma_{Mut,\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\}}$ and $p_{Mut,\{\mathbb{B},\mathbb{S}\}}$. These variables are also recombined from the parent individuals so that the *mutation strength* is adapted over time as well. This is achieved in a similar way compared to the numeric variable types using the fitness-weighted averages

$$\sigma_{Mut,\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\},recom} = \frac{1}{\sum_{k=1}^{\rho} \mathbf{fit}_{rel}(j_k)} \sum_{k=1}^{\rho} \mathbf{fit}_{rel}(j_k) \cdot \sigma_{Mut,\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\}}^{(j_k)} \tag{5.33}$$

and

$$p_{Mut,\{\mathbb{B},\mathbb{S}\},recom} = \frac{1}{\sum_{k=1}^{\rho} \mathbf{fit}_{rel}(j_k)} \sum_{k=1}^{\rho} \mathbf{fit}_{rel}(j_k) \cdot p_{Mut,\{\mathbb{B},\mathbb{S}\}}^{(j_k)} \tag{5.34}$$

in which $\sigma_{Mut,\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\}}^{(j_k)}$ and $p_{Mut,\{\mathbb{B},\mathbb{S}\}}^{(j_k)}$ denote the mutation parameter of the j_kth individual in P_t.

Mutation Operator

The mutation operator is applied to all recombined individuals and serves as an important method to explore new areas of the search space. The standard Evolution Strategies use a continuous, numerical vector space and define the mutation operator as an random "noise" vector that is added to the recombined vectors. The noise vector is modeled using a multivariate normal

distribution $\mathcal{N}(\mathbf{0}, \Sigma)$ with a zero mean vector and a covariance matrix Σ. The covariance matrix describes correlations between the optimization variables, e.g., if variable v_1 is increased, v_2 should also be increased. The correlation or dependency of variables in the mutation operator is beneficial because it allows a more precise "steering" of the search process. When two variables v_1 and v_2 are positively[6] correlated, a mutation operator that considers the correlation would generate more individuals in which v_1 and v_2 are increasing (or decreasing) together. At least in theory, more promising solutions would be generated with a higher probability.

The concept of variable correlation in the mutation operator of the extended Evolution Strategies would certainly be desirable as well. However, the proposed extended Evolution Strategies use five heterogeneous variable types with discrete and categorical types. A useful definition of correlation between numerical and categorical variables is not possible in a straightforward way. Furthermore, a covariance matrix is square and would have N_{Genes}^2 entries. As the number of variables is expected to easily become larger than 1,000, especially due to the feature selection, this matrix would contain millions of entries. In order to prevent numerical issues and unnecessary computations, the mutation of each variable type is independently defined as follows:

- All of the three numerical variable types are mutated using an additive, normally distributed random variable – similarly to the standard Evolution Strategies. The mutation parameter $\sigma_{Mut,\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\}}$ determines the one-dimensional standard deviation of the normal distribution. It is initialized individually for each variable using

$$\sigma_{Mut,\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\},init} = \chi_{Mut} \cdot \left(v^{max}_{\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\}} - v^{min}_{\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\}}\right) \tag{5.35}$$

  in which $\chi_{Mut} \in [0, 1] \subset \mathbb{R}$ connects the initial mutation strength with the value range. The metaparameter is empirically determined to $\chi_{Mut} = 0.2$ which sets the initial standard deviation of the mutation to 20% of the corresponding variable's value range. The mutated value is obtained by

$$\tilde{v}_{\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\},mut} = v_{\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\},recom} + \mathcal{N}\left(0, \sigma_{Mut,\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\}}\right) \tag{5.36}$$

  in which the mutation parameter $\sigma_{Mut,\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\}}$ determines the standard deviation of the normal distribution. The mutation can produce values that lie outside of the variable constraints. Therefore, the minimum and maximum limits are obeyed using

$$v_{\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\},mut} = \min\left(\max\left(\tilde{v}_{\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\},mut}, v^{min}_{\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\}}\right), v^{max}_{\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\}}\right). \tag{5.37}$$

[6] A positive correlation between two variables v_1 and v_2 can be expressed as "v_1 increases when v_2 is increased" or "v_1 decreases when v_2 is decreased". A negative correlation would be "v_1 decreases when v_2 is increased" or "v_1 increases when v_2 is decreased".

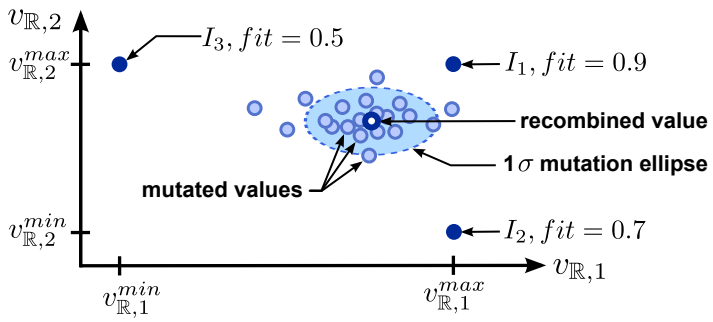


Figure 5.8: Visualization of the effect of the recombination and mutation operator for two numerical hyperparameters. In this example, three parent individuals with different fitness values are recombined according to their fitness-weighted average properties. This leads to a recombined value that is close to I_1 with the highest fitness. Some mutated individuals are shown to indicate the typical variation of the mutation operator.

Additionally, the value is rounded afterwards for the integer variable type.

- Boolean variables are mutated using a random bit flip whose probability is controlled by the corresponding mutation parameter $p_{Mut,\mathbb{B}}$. This metaparameter is initialized with $p_{Mut,\mathbb{B},init} = 0.1$. Let $rand \sim \mathcal{U}_{\mathbb{R}}(0,1)$ be a random number so that

$$v_{\mathbb{B},mut} = \begin{cases} \neg v_{\mathbb{B},recom} : & rand \leq p_{Mut,\mathbb{B}} \\ v_{\mathbb{B},recom} : & else \end{cases} . \tag{5.38}$$

- Categorical variables are handled similarly to Boolean ones. The mutation parameter $p_{Mut,\mathbb{S}}$ is initialized with $p_{Mut,\mathbb{S},init} = 0.2$. In contrast to the Boolean variables, the mutation is realized by the selection of a random item out of the base set

$$v_{\mathbb{S},mut} = \begin{cases} randomItem(S_{\mathbb{S}}) : & rand \leq p_{Mut,\mathbb{S}} \\ v_{\mathbb{S},recom} : & else \end{cases} \tag{5.39}$$

in which $rand \sim \mathcal{U}_{\mathbb{R}}(0,1)$.

Figure 5.8 shows the effect of the combination of the recombination and mutation process.

Self-Adaptive Mutation Strength

The *mutation strength* is controlled by the mutation parameters of each variable. Numerical variables use the standard deviations of a normal distribution

$\sigma_{Mut,\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\}}$ to control the mutation strength – the higher the standard deviations are, the higher the impact of the mutation will be. Boolean and categorical variables use probabilities $p_{Mut,\{\mathbb{B},\mathbb{S}\}}$ that control random selections. The higher these probability values are, the higher the mutation strength becomes.

The mutation strength has a great impact on the success of the whole algorithm. A too strong mutation may lead to divergence, while a too small mutation slows down the optimization and can lead to the termination of the algorithm in local optima. The best amount of mutation strength is usually not known before the optimization. Therefore, the dynamic adaptation of the mutation parameters during the optimization is a common concept in Evolution Strategies. A very early variant is the so-called 1/5 success rule which controls the mutation strength according to the success of mutations. The idea is that the success rate of the offspring generation – in terms of fitness improvements compared to the currently best solution – should be equal to 1/5. However, this rule was developed and tested for very simple, low-dimensional and almost noise-free objective functions and performs poorly for more complex functions. The current state-of-the-art mutation adaptation for continuous numerical optimization is the CMA-Evolution Strategies variant, the *Covariance Matrix Adaptation* [Hansen, 2006]. The covariance matrix is adapted during the optimization using statistical properties of the fitness distribution. However, the CMA approach is not applicable in the proposed extended Evolution Strategies as there is no mutation covariance matrix due to the heterogeneous variable types (see the previous paragraph).

Instead, the so-called *extended log-normal* variant of self-adaptation of uncorrelated mutation parameters [Beyer and Schwefel, 2002] is used in the extended Evolution Strategies. The mutation parameters of all variables are evolved over time using a mutative approach that incorporates a global and a local mutation strength variation. For each generation, a global mutation adaptation $rand_{global} \sim \mathcal{N}(0,1)$ is determined. Then another local random variable $rand_{local} \sim \mathcal{N}(0,1)$ is drawn for each individual and each variable in the genotype $\mathbb{G}$, which is used to mutate the recombined mutation parameters

$$\sigma_{Mut,\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\},mut} = \sigma_{Mut,\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\},recom} \cdot \exp(\tau_1 \cdot rand_{global} + \tau_2 \cdot rand_{local}) \tag{5.40}$$

and

$$p_{Mut,\{\mathbb{B},\mathbb{S}\},mut} = p_{Mut,\{\mathbb{B},\mathbb{S}\},recom} \cdot \exp(\tau_1 \cdot rand_{global} + \tau_2 \cdot rand_{local})\,. \tag{5.41}$$

The exponential function $\exp(x)$ is used as a factor that is smaller than one (but always greater than zero) for $x < 0$ and larger than one for $x > 0$. For instance, $\exp(0.5) \approx 1.65$ leads to an increased mutation strength of 65 percentage points and $\exp(-1.2) \approx 0.30$ decreases the mutation strength by

around 70 percentage points. The exponent depends on a linear combination of global and local adaptation which are controlled by two factors τ_1 and τ_2. The local adaptation ensures that each mutation parameter is adapted independently. However, the chance to significantly increase (or decrease) all mutation parameters at the same time is nearly zero, even though this is very helpful to quickly escape from local optima. Therefore, the global adaptation is used to change all mutation variables to become larger or smaller at the same time. The weight between global and local mutation adaptation is chosen to be balanced equally in the extended Evolution Strategies – leading to $\tau_1 = 0.5$ and $\tau_2 = 0.5$.

Termination Criteria

The termination of an Evolutionary Algorithm is a crucial aspect because on the one hand, a fast optimization speed is desired, but on the other hand, a premature stopping of the algorithm increases the chance to find one of numerous local optima. The goal is to find a compromise between runtime and result quality.

The proposed extended Evolution Strategies use a termination criterion based on a minimum fitness improvement threshold. The generation index $0 \leq t \leq N_{Generations}$ is increased after every generation, however, the actual number of generations $N_{Generations}$ is not known before. In the tth generation the best fitness of the current population is determined with $fit_{best}(t) = \max(\mathbf{fit}(P_t))$. Furthermore, the overall best fitness value that has been observed so far

$$fit_{best,overall}(t) = \max(fit_{best}(0), fit_{best}(1), \ldots, fit_{best}(t)) \tag{5.42}$$

is calculated. The improvement of the overall fitness during the last Δ_t generations is calculated using

$$fitImprovement(t) = fit_{best,overall}(t) - fit_{best,overall}(t - \Delta_t) \tag{5.43}$$

with $\Delta_t \leq t$. Note that at least the initial and Δ_t further generations need to be evaluated. The fitness improvement function is one part of the termination criteria

$$term(t) = \begin{cases} 1: & fitImprovement(t) < \epsilon_{fit} \ \wedge \ t \geq N_{Generations,min} \\ 0: & else \end{cases}. \tag{5.44}$$

The threshold is triggered when a minimum fitness improvement of ϵ_{fit} is not reached anymore. Additionally, a minimum number of generations

$$N_{Generations,min} \geq \Delta_t \tag{5.45}$$

is defined that prevents a too early termination. The values of Δ_t, ϵ_{fit} and $N_{Generations,min}$ have to be carefully chosen for each application (see section 5.5.3).

5.5 Evolutionary Configuration Adaptation

The previous section presents the extended Evolution Strategies, which are the foundations of the proposed *Evolutionary Configuration Adaptation* (*ECA*) algorithm to solve the configuration adaptation problem. The goal of the *ECA* algorithm is the simultaneous optimization of all components of the classification pipeline configuration. It is well studied in literature (see section 3.2.4) that Evolutionary Algorithms are able to solve parts of the configuration adaptation problem. However, the full problem is more complex, especially because of the hierarchical structure of the hyperparameters that depend on the choice of the selected algorithms.

The *ECA* algorithm tackles the hyperparameter dependency problem with the introduction of a hypothetical "master" machine learning algorithm that includes the combined set of *all* hyperparameters – of all feature transforms and all classifiers. This approach can be understood as a linearization of the hierarchical problem so that it fits to the linear structure of the genotype $\mathbb{G}$. This combined set of all hyperparameters is expected to become relatively large – the complexity analysis of the configuration adaptation problem (see section 5.2.2) roughly assumes a total number of more than 40 hyperparameters. However, obviously, not all hyperparameters have an effect on the classification pipeline *simultaneously*. When a pipeline configuration is derived from an individual (see section 5.5.2) the selection of the algorithms acts as a "switch": The set of hyperparameters that belongs to the selected algorithms is activated while the others are ignored.

This way of hyperparameter handling can also be motivated with the functioning of natural genetics and the *genotype-phenotype distinction* [Johannsen, 1911]. The genotype contains all genetic information while only parts[7] of it influence the phenotype, i.e. the set of actual physical properties of the organism. This means that parts of the genotype remain "silent". The hyperparameter handling of the *ECA* algorithm shows similarities to this biological principle. The genotype contains numerous hyperparameters of which a large part is inactive most of the time and does not influence the actual classification pipeline – the phenotype. However, the information of all hyperparameters is inherited regardless of their activation state, which is also the case for the silent genes in the DNA of living organisms.

Another interesting and potentially useful side effect of this approach is that one individual contains information about multiple, but partly similar configurations. The selection of, e.g., a different classifier activates an alternative configuration with corresponding hyperparameters, which is optimized henceforth. However, the information of the hyperparameter values of

[7]The actual physical properties are not only determined by the genetic information but also influenced by the environment – known as *gene-environment interplay* [Rutter, 2010].

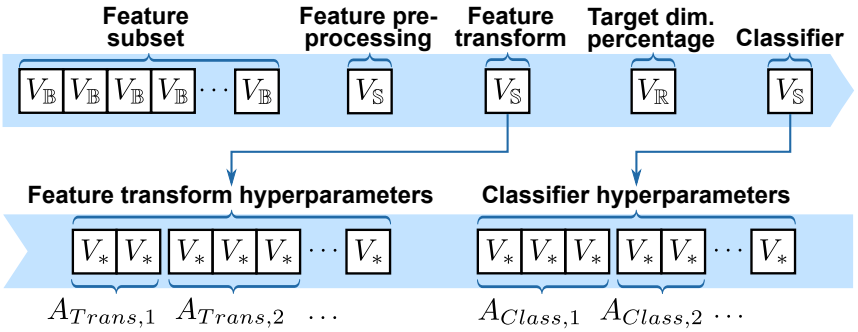


Figure 5.9: Genotype structure of the *ECA* optimization algorithm. The feature transform and classifier selection act as switches that activate the corresponding set of hyperparameters. Note that the genotype is a linear list of variables and is only separated into two parts here for visualization purposes.

the previously selected classifier also remains in the genotype and could be activated and tuned again in later generations.

A negative effect of this approach is that it comes with a slightly higher computational cost as all evolutionary operators are applied on parts of the genotype that do not contribute directly to the configuration. However, the computational effort of the evolutionary operators is usually negligible compared to the fitness evaluation of the individuals, which comprises the training of complex machine learning algorithms.

5.5.1 Genotype Coding

The genotype $\mathbb{G}_{ECA}$ of the *ECA* algorithm contains all parts of the pipeline configuration θ (see section 4.6). Figure 5.9 shows the structure of the genotype which is defined as follows:

1. The feature subset $S_{FeatSubSet}$ is coded as a sequence – or bitstring – of D_{in} Boolean variables

$$V_{FeatSel} = [V_{\mathbb{B},1}, V_{\mathbb{B},2}, \ldots, V_{\mathbb{B},D_{in}}]\,. \tag{5.46}$$

   The initial probability values $p_{Init,\mathbb{B},j}$ are used to incorporate prior knowledge of the feature relevance into the optimization process (see section 5.5.4).

2. The feature preprocessing method $A_{PreProc}$ is coded as a single categorical variable $V_{\mathbb{S}}$. The corresponding base set $S_{\mathbb{S}}$ is equal to the portfolio of feature preprocessing methods $S_{PreProc}$ (see section 4.5.2 and appendix B).

3. The feature transform method A_{Trans} is coded as a single categorical variable $V_{\mathbb{S}}$. The corresponding base set $S_{\mathbb{S}}$ is equal to the portfolio of feature transform methods S_{Trans} (see section 4.5.3 and appendix C).

4. The target dimensionality D_{Trans} is a common "hyperparameter" of almost all feature transforms and it is assumed that it depends on the intrinsic dimensionality of the dataset. Therefore, D_{Trans} is added outside of the set of hyperparameters that depends on specific feature transform algorithms. However, the target dimensionality is connected to the number of selected features and

$$1 \leq D_{Trans} \leq D_{FeatSel} = |S_{FeatSubSet}| \,. \tag{5.47}$$

Therefore, the concept of *target dimensionality percentages* is introduced with

$$0 \leq \delta_{TargetDimPerc} \leq \delta_{TargetDimPerc,Max} \leq 100 \,. \tag{5.48}$$

The value $\delta_{TargetDimPerc} \in \mathbb{R}$ denotes the percentage of the number of features that should be used from the selected ones

$$D_{Trans} = \max\left(\text{round}\left(\frac{\delta_{TargetDimPerc}}{100} \cdot D_{FeatSel}\right), 1\right). \tag{5.49}$$

The upper constraint $\delta_{TargetDimPerc,Max} \in \mathbb{R}$ limits the maximum percentage which is introduced to incorporate prior knowledge about dimensionality reduction into the optimization algorithm: A rather strong dimensionality reduction is desired to circumvent the curse of dimensionality. Consider the case of $D_{in} = 1{,}000$ which could lead to $D_{Trans} = 1{,}000$ in case of $\delta_{TargetDimPerc} = 100$. This would likely produce computationally expensive and counterproductive transformations without any dimensionality reduction effect. Therefore, the idea is to limit the maximum target dimensionality. Empirical studies have revealed that $D_{Trans,Max} = 50$ is a reasonable maximum dimensionality. The actual dimensionality limitation is achieved by

$$\delta_{TargetDimPerc,Max} = \min\left(100, 100 \cdot \frac{D_{Trans,Max}}{D_{in}}\right) \tag{5.50}$$

which takes the ratio of $D_{Trans,Max} \in \mathbb{N}$ and the input dimensionality into account. Figure 5.10 shows the resulting relation between the upper constraint $\delta_{TargetDimPerc,Max}$ and D_{in}. The value of $\delta_{TargetDimPerc,Max}$ decreases for $D_{in} > D_{Trans,Max}$.

Finally, the target dimensionality percentage $\delta_{TargetDimPerc}$ is coded as linearly scaled, real-valued variable $V_{\mathbb{R}}$ into the genotype with the value constraints of $v_{\mathbb{R}}^{min} = 0$ and $v_{\mathbb{R}}^{max} = \delta_{TargetDimPerc,Max}$.

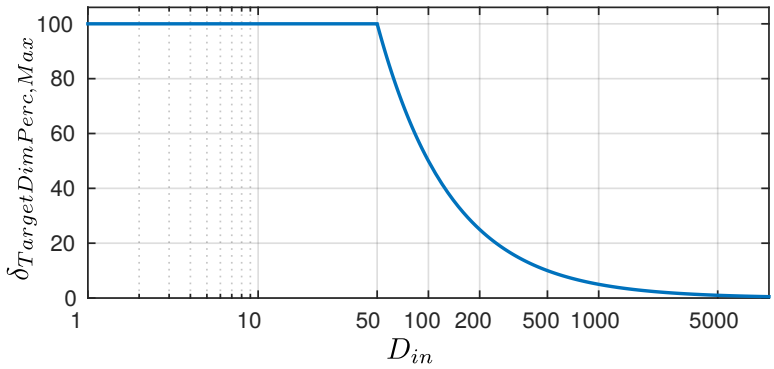


Figure 5.10: Restrictions of the maximum target dimensionality percentage depending on the input dimensionality D_{in}. This restriction forces dimensionality reduction with a target dimensionality of at most $D_{Trans} = 50$.

5. The classifier A_{Class} is coded as a single categorical variable $V_{\mathbb{S}}$. The corresponding base set $S_{\mathbb{S}}$ is equal to the portfolio of classifiers $S_{Classifiers}$ (see section 4.5.4 and appendix D).

In addition to that, all hyperparameters of all selectable algorithms of the feature transform portfolio S_{Trans} and the classifier portfolio $S_{Classifiers}$ are appended to the genotype. Note that the portfolio of preprocessing algorithms $S_{PreProc}$ does not contain any method with hyperparameters and therefore, no further hyperparameters have to be considered[8].

The hyperparameter appending process is the same for both portfolios and works the following way. For each algorithm $A_l \in S_{Portfolio}$ in a portfolio the set of corresponding hyperparameters

$$S_{\mathbb{H},A_l} = \{H_{l,1}, H_{l,2}, \ldots, H_{l,N_{Hyp}(A_l)}\} \quad \text{with} \quad H_j \in \{H_{\mathbb{R}}, H_{\mathbb{Z}}, H_{\mathbb{S}}\} \tag{5.51}$$

is determined according to section 3.2.4. The three basic types of hyperparameters can directly be mapped to corresponding variable types of the extended Evolution Strategies (see section 5.4.1):

- *Continuous, real-valued hyperparameters* are mapped depending on the type of the hyperparameter. "Normal", real-valued hyperparameters are mapped to linearly scaled, real-valued variables using

$$H_{\mathbb{R}} \mapsto V_{\mathbb{R}} \quad \text{with} \quad v_{\mathbb{R}}^{min} = h_{\mathbb{R}}^{min} \;,\; v_{\mathbb{R}}^{max} = h_{\mathbb{R}}^{max} \,. \tag{5.52}$$

  Real-valued hyperparameters with an exponentially scaled value range such as Gaussian kernel hyperparameters are mapped as exponentially scaled, real-valued variables

$$H_{\mathbb{R}} \mapsto V_{exp\mathbb{R}} \quad \text{with} \quad v_{exp\mathbb{R}}^{min} = \log_{10}(h_{\mathbb{R}}^{min}) \;,\; v_{exp\mathbb{R}}^{max} = \log_{10}(h_{\mathbb{R}}^{max}) \,. \tag{5.53}$$

[8]It would easily be possible to consider hyperparameters of preprocessing algorithms as well by linearly appending them to the genotype like the other hyperparameters.

Note that $h_{\mathbb{R}}^{min} > 0$ and $h_{\mathbb{R}}^{max} > 0$ so that the logarithm values are always well defined.

- *Integer hyperparameters* are directly mapped using

$$H_{\mathbb{Z}} \mapsto V_{\mathbb{Z}} \quad \text{with} \quad v_{\mathbb{Z}}^{min} = h_{\mathbb{Z}}^{min} \;,\; v_{\mathbb{Z}}^{max} = h_{\mathbb{Z}}^{max} \,. \tag{5.54}$$

- *Categorical hyperparameters* are mapped using

$$H_{\mathbb{S}} \mapsto V_{\mathbb{S}} \quad \text{with} \quad S_{\mathbb{S}} = S_{Cat} \,. \tag{5.55}$$

The variable information – such as the value ranges of numerical hyperparameters and the base sets for categorical hyperparameters – is set according to reasonable standard values, which can be found in appendix C for the feature transforms and in appendix D for the classifiers. The corresponding mutation parameter $\sigma_{Mut,\{\mathbb{R},exp\mathbb{R},\mathbb{Z}\}}$ or $p_{Mut,\mathbb{S}}$ of each variable is set as described in the subsection about the mutation operator in section 5.4.2. Finally, the resulting variable $V_* \in \{V_{\mathbb{R}}, V_{exp\mathbb{R}}, V_{\mathbb{Z}}, V_{\mathbb{S}}\}$ for each hyperparameter is appended to the genotype $\mathbb{G}_{ECA}$.

The aforementioned hyperparameter switch principle requires the storage of the connection between the algorithms and the positions of the corresponding hyperparameters in the genotype $\mathbb{G}_{ECA}$. This is realized with a *hyperparameter genotype mapping* (HGM), which stores the indices $j_{\mathbb{G}}$ of the variables $V_{*,j_{\mathbb{G}}}$ in the genotype that represent the kth hyperparameter of the lth algorithm. The mapping function for one algorithm portfolio is formally defined as

$$HGM : (l, k) \mapsto j_{\mathbb{G}} \,. \tag{5.56}$$

Consequently, two of these mapping functions $\{HGM_{Trans}, HGM_{Class}\}$ are necessary for the hyperparameters of the feature transform and the classifier, respectively.

5.5.2 Configuration Generation and Fitness Evaluation

An individual $I_{ECA} \in \mathbb{G}_{ECA}$ contains all variables in the genotype to almost directly obtain a fully specified pipeline configuration $\theta_{I_{ECA}}$ (see section 4.6) that is the phenotype. The subset $S_{FeatSubSet}$ contains index numbers that are obtained depending on the activated feature selection variables in the bitstring $V_{FeatSel}$

$$j \in S_{FeatSubSet} \quad \Leftrightarrow \quad v_{\mathbb{B},j} = 1 \tag{5.57}$$

in which $v_{\mathbb{B},j}$ is the value of the jth Boolean variable in $V_{FeatSel}$ and $1 \leq j \leq D_{in}$. The choices of the feature preprocessing method $A_{PreProc}$, the feature transform A_{Trans} and classifier algorithm A_{Class} are coded directly in I_{ECA}. The target dimensionality is coded in I_{ECA} as percentage $\delta_{TargetDimPerc}$

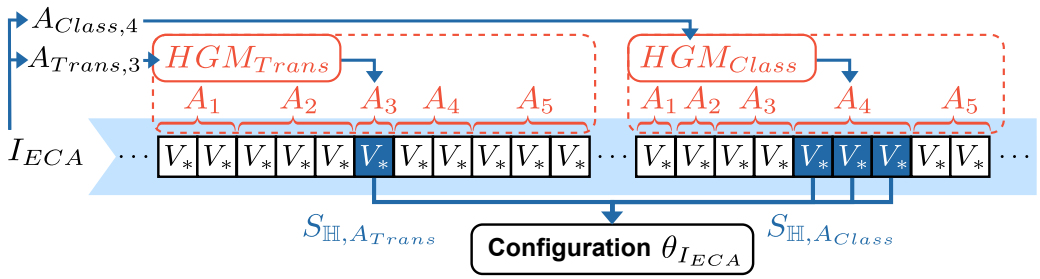


Figure 5.11: Principle of the hyperparameter selection from the genotype to a valid pipeline configuration – the phenotype. The third feature transform and the fourth classifier are chosen in this example and only the corresponding hyperparameters are used in the configuration $\theta_{I_{ECA}}$.

of the number of selected features $D_{FeatSel} = |S_{FeatSubSet}|$ while the final dimensionality D_{Trans} is obtained according to equation 5.49. The individual I_{ECA} contains all hyperparameters for all algorithms and therefore, the correct subset has to be chosen, depending on the selection of the feature transform and the classifier. This hyperparameter selection requires the hyperparameter genotype mapping and is done by

$$hyperSel_{Trans}(I_{ECA}, HGM_{Trans}) = S_{\mathbb{H},A_{Trans}} \tag{5.58}$$

and

$$hyperSel_{Class}(I_{ECA}, HGM_{Class}) = S_{\mathbb{H},A_{Class}}\,. \tag{5.59}$$

The hyperparameter selection principle is depicted in figure 5.11. Finally, the fitness of the resulting configuration $\theta_{I_{ECA}}$ is determined using the holistic cross-validation (see section 5.1.2)

$$fit(I_{ECA}) = holisticCV(\theta_{I_{ECA}}, T_{Train}) \tag{5.60}$$

and is assigned to the individual I_{ECA}.

5.5.3 Metaparameters of the ECA Algorithm

One disadvantage of Evolutionary Algorithms is the relatively large number of metaparameters that influence the optimization process. There is unfortunately no general recipe to choose suitable metaparameter values for a specific task. On the one hand, the size of the search space of the configuration adaptation problem is very huge and bears many local optima in the objective function. Therefore, a rather large initial population μ_{init} and a large number of children λ in each generation would be advantageous. On the other hand, the fitness evaluation function is computationally expensive due to the holistic cross-validation method. Consequently, a compromise between optimization runtime and quality has to be made. The following metaparameters

for the extended Evolution Strategies are determined on the basis of preliminary empirical studies.

A large number of, e.g., more than thousand initial individuals would definitely be wishful, but the early discarding system in the holistic cross-validation (see section 5.1.3) is not very effective in the initial phase. The reason is that the best cross-validation accuracy values are rather low at the beginning and thus not many configurations are discarded early enough. Therefore, the initial population size is limited to $\mu_{init} = 400$ and $\lambda = 200$ children are generated in every generation. The number of individuals that survive each generation is set to $\mu = 20$, which is a compromise between selection pressure and diversity. The number of parents is determined to $\rho = 3$ so that a reasonable amount of genetic variation is introduced to the offspring generation. The total lifespan of individuals is limited to $\kappa = 4$ generations so that new solutions have the chance to evolve and "take the lead" after some generations.

The metaparameters for the termination criteria are selected as follows. The minimum fitness improvement after $\Delta_t = 3$ consecutive generations is determined as $\epsilon_{fit} = 0.001$. This means that at least an improvement of 0.1 percentage points of cross-validation accuracy is required to continue with the evolution process. Furthermore, the minimum number of generations is set to $N_{Generations,min} = 5$.

In addition to these evolutionary metaparameters discussed in this section, the *ECA* algorithm has more metaparameters that are described in this and other chapters. The full list of metaparameters can be found in appendix E. Note that all of these metaparameters are *not* optimized within the actual optimization process. However, empirical investigations have shown that the proposed default values lead to reasonable performance results for a wide range of learning tasks.

5.5.4 Improvement of the Initial Population

The improvement of the initial population is a standard approach in Evolutionary Algorithms to improve the optimization speed and to decrease the probability of getting stuck in local optima. If the initial population already contains at least one good solution, the actual evolutionary optimization only needs to perform a "fine-tuning". However, prior knowledge of the problem is needed to obtain such solutions that perform better than random ones.

Problem relaxation is one way to obtain prior knowledge: a simplified problem approximation is considered that can be solved easier and faster. Parts of the solutions of the problem approximation are then used to obtain an improved initial population. The feature selection problem contributes the most to the overall problem complexity (see section 5.2.2) and therefore, it is the most promising starting point to improve the population. The use

of a filter approach only considers the feature selection problem and delivers information about the usefulness of single features. The random forest variable importance metric proposed by [Genuer et al., 2010] (see section 3.2.2) is especially suitable to predict the feature usefulness in a fast way. Usually after only a few seconds, this approach returns a real-valued metric $f_{FeatImp}(j) \in [0,1] \subset \mathbb{R}$ for each feature $1 \leq j \leq D_{in}$.

An elegant way to fuse this information to the *ECA* algorithm is the adaptation of the initialization probability values of the Boolean variables $V_{FeatSel}$ for the feature selection problem. The idea is that the more important a feature is, the more likely it should be selected in the random population initialization mode. However, the importance metric value $f_{FeatImp}(j)$ should not be directly used as the initialization probability of the jth feature $p_{Init,\mathbb{B},j}$ because there would be a too large influence of the characteristics of the random forest on the feature selection. This metric – as filter-based feature selection methods in general – does not consider any feature interactions or the influence of the feature transform. Therefore, the importance metric values are transformed to a range of $[p_{Feat,min}, 1] \subset \mathbb{R}$ so that a minimum probability value $p_{Feat,min} > 0$ can be determined. This is achieved by

$$p_{Init,\mathbb{B},j} = p_{Feat,min} + (1 - p_{Feat,min}) \cdot f_{FeatImp}(j) \quad \in \ [p_{Feat,min}, 1] \subset \mathbb{R} \tag{5.61}$$

and so even the lowest importance metric values – which would be zero – still lead to a probability greater than zero to select that feature. This probability value is set to $p_{Feat,min} = 0.25$ so that still a reasonable chance of 25% remains for features that are considered as absolutely unimportant by the random forest. It is expected that the optimization process is not very sensitive to specific values of $p_{Feat,min}$.

5.5.5 Optimization Trajectory

During the optimization many different configurations are evaluated with the holistic cross-validation target metric. In order to keep track of the optimization trajectory, each of the $N_{Configs}$ configurations is stored together with the corresponding fitness value in a set of tuples

$$\begin{aligned} optTrajectory = \{&(\theta_1, holisticCV(\theta_1, T_{Train}), \\ &(\theta_2, holisticCV(\theta_2, T_{Train}), \ldots, \\ &(\theta_{N_{Configs}}, holisticCV(\theta_{N_{Configs}}, T_{Train})\}. \end{aligned} \tag{5.62}$$

After the *ECA* algorithm terminates, the optimization trajectory is ranked by the fitness values so that a function $rankedConfigs(l_{rank})$ returns the l_{rank}th best or "fittest" configuration with $1 \leq l_{rank} \leq N_{Configs}$. In case multiple configurations have the same fitness value, the earliest evaluated

configuration is returned. The function $rankedConfigsFit(l_{rank})$ returns the corresponding fitness values of the l_{rank}th configuration.

Obviously, the overall best performing configuration $rankedConfigs(1)$ is the most interesting result to be used for setting up a classification pipeline for a real application. However, the optimization trajectory contains more information that is exploited with extended analyses in chapter 6.

5.6 Variants of the ECA Algorithm

The number of degrees of freedom of the classification pipeline is large and there are basically five components that contribute to this adaptability: The feature selection, the three algorithm portfolios of the feature preprocessing, the feature transform as well as the classifier and finally the hyperparameter adaptation. On the one hand, the study of solutions for machine learning challenges (see chapter 3) leads to the expectation that all of these aspects are potentially helpful to obtain a good and well generalizing classification pipeline. The proposed classification pipeline with the *ECA* optimization algorithm is able to handle all of the five components at the same time. Therefore, it provides the most comprehensive amount of adaptability which is, of course, the most interesting variant and will be denoted as *ECA-full* algorithm in the following.

But on the other hand, there are multiple, non-negligible risks of the huge adaptability: First, the risk for any heuristic optimization algorithm of getting stuck in local optima is large. Secondly, the risk of overfitting to the training dataset leading to a poor generalization performance is evident. And thirdly, an unreasonable amount of computation time could be spent with few or even no measurable benefit.

In order to contribute to the understanding of the internal functionality of the AROMS-Framework, several variants of the *ECA* algorithm with relaxations of the configuration adaptation problem are analyzed. In these "easier" optimization problems less degrees of freedom are considered, which is achieved by removing or restricting certain components of the classification pipeline. The consideration of all possible combinations of the five main components (see above) being either optimized or not would lead to a total number of $2^5 = 32$ possible variants of the *ECA* algorithm. The analysis of all these combinations would be an unreasonable amount of comparisons with few insights.

Therefore, the leave-one-out principle is applied to quantify the contribution of each of the main components. This approach is similar to the idea of the random forest variable importance metric (see section 3.2.2): a component is considered as important, if the performance drops significantly when it is missing. This idea leads to five variants of the *ECA* algorithm:

Optimization variant	Feature selection	Full feature preprocessing portfolio	Full feature transform portfolio	Full classifier portfolio	Hyperparameter tuning
ECA-full	✓	✓	✓	✓	✓
ECA-noFeatSel	–	✓	✓	✓	✓
ECA-noPreProc	✓	–	✓	✓	✓
ECA-noTrans	✓	✓	–	✓	✓
ECA-simpleClassifier	✓	✓	✓	–	✓
ECA-defaultHyper	✓	✓	✓	✓	–

Table 5.1: Overview of the variants of the *ECA* algorithm and the corresponding components that are optimized. The symbol "✓" means that the component is optimized, while "–" indicates that the component is not optimized.

- The *ECA-noFeatSel* variant leaves out the feature selection problem and always selects all features so that $S_{FeatSubSet} = \{1, 2, \ldots, D_{in}\}$ and $|S_{FeatSubSet}| = D_{in}$.
- The *ECA-noPreProc* variant limits the feature preprocessing portfolio $S_{PreProc}$ to the identity function.
- The *ECA-noTrans* variant limits the feature transform portfolio S_{Trans} to the identity function.
- The *ECA-simpleClassifier* variant limits the classifier portfolio $S_{Classifiers}$ to the naïve Bayes classifier, which is a simple classifier without any hyperparameters.
- The *ECA-defaultHyper* variant does not consider hyperparameter tuning and uses the corresponding default values[9].

Table 5.1 provides an overview of the proposed *ECA* variants and their corresponding properties. However, it is likely the case that the impact of specific pipeline components is highly dependent on the dataset. It might be the case that other components compensate a missing one, e.g., the feature transform can also extract relevant features when no feature selection is performed. This does not generally mean that feature selection is not a useful pipeline component. Instead, multiple datasets have to be considered to achieve statistically relevant insights.

[9]The appendices C and D list these default values for feature transforms and classifiers, respectively. The default values are proposed by the implementations of the corresponding methods.

Chapter 6

Extended Optimization Analyses

The previous chapter presents the *ECA* optimization algorithm to find suitable pipeline configurations for a given learning task. After each relatively time-consuming optimization process it is possible to exclusively use the best configuration with the highest fitness. A single configuration is sufficient to generate a ready-to-use classification pipeline.

However, the optimization trajectory – a set of configurations and corresponding fitness values – also contains useful information about the learning task and possible solutions. The distribution of the best configurations is of special interest as it can be expected that multiple, well performing configurations will be found. This chapter presents extensions to the AROMS-Framework that exploit the generated data during the optimization. The proposed extended analyses tackle two aspects: First, the gain of knowledge about the classification problem for the user and secondly, the improvement of the generalization performance. A case study with an image-based object recognition task is used to demonstrate the proposed extensions on realistic data.

This chapter is organized as follows. In section 6.1 the case study dataset is introduced and the "raw" results of the *ECA* optimization algorithm are described. The first extension comprises meaningful visualizations of the best configurations that are presented in section 6.2. The second extension is a multi-pipeline classifier of the best configurations that is proposed in section 6.3. Finally, a discussion of the extensions can be found in section 6.4.

6.1 Case Study Dataset

The proposed extensions rely on the data that is generated during the optimization process. Therefore, a realistic case study dataset and the corresponding results of the optimization algorithm are useful to motivate the

Figure 6.1: Example images from the *coins* dataset for the case study. The image shows three coins of each class with 1-, 2-, 5-, 10- and 20-cent coins from the left to the right side.

methodologies. On the one hand, the dataset should be realistic and "difficult" enough to show the benefits of the AROMS-Framework compared to standard classifiers. On the other hand, the classification problem and its results should still be interpretable to verify the usefulness of the proposed extensions. In the following a suitable case study dataset is presented. After that, the *ECA* optimization algorithm is applied to this task and the results are presented. These serve as the initial point for the extended analyses.

6.1.1 Dataset Description

The case study dataset consists of an image-based object recognition problem with typical properties that machine learning experts are regularly faced with. The task of the *coins* dataset is the distinction of euro coins based on color images and is also used in [Bürger and Pauli, 2015a]. The dataset comprises five different classes, namely 1-, 2-, 5-, 10- and 20-cent coins with 64 color images for each class, leading to 320 images in total. All coins are captured with the front side up, however, the coin rotation angle is arbitrary. Furthermore, each coin is segmented from a white background so the data acquisition conditions can be considered as relatively well controlled. Figure 6.1 shows example images of the dataset. The intra-class variance is relatively high due to the large variation in shading, reflection and color properties.

In order to classify the images, a standard feature-based approach is chosen. The challenge is that even an expert cannot "guess" the optimal feature set for the given task. It is a common practice to use a set of popular and promising standard features according to intuitive properties of the problem and the data. For instance, the object size should be considered since the physical coin sizes are different. The color information is expected to be important to distinguish the copper-colored 1-, 2- and 5-cent coins from the gold-colored 10- and 20-cent coins. Furthermore, several rotation-invariant

texture descriptors are potentially useful. However, it can be expected that no single feature alone can adequately solve the five-class problem. The case study uses the following feature groups (see section 4.4) that comprise features which are explained and referenced in appendix A:

- object area in pixels (1 feature group with 1 dimension),
- statistical features of the gray value histogram: mean and standard deviation (2 feature groups with 1 dimension each),
- statistical features of the color/hue channel histogram of the image: mean and standard deviation (2 feature groups with 1 dimension each),
- Hu moments of the gray value texture (1 feature group with 7 dimensions),
- Local Binary Patterns (LBP) gray value texture descriptors in different variants: uniform (1 feature group with 59 dimensions), rotation invariant (1 feature group with 36 dimensions) as well as uniform + rotation invariant (1 feature group with 10 dimensions) and
- low-level pixel features in form of down-scaled gray value images: 5×5 pixels (1 feature group with 25 dimensions), 10×10 pixels (1 feature group with 100 dimensions) and 20×20 pixels (1 feature group with 400 dimensions).

In total, there are 12 feature groups and in case all feature groups are concatenated a total feature space dimensionality of $D_{in} = 642$ dimensions is obtained. This leads to a sample to feature ratio of $v_{SFR} = 320/642 = 0.498$ that can be considered as suboptimal regarding the curse of dimensionality. However, this is typical for image-based classification tasks in which labeled ground truth data is expensive.

The full dataset of 320 coins is randomly subdivided into 50% training and 50% test dataset, however, this division is fixed for all experiments. Any learning algorithm may only use the training dataset to adapt its model parameters or tune its hyperparameters. The *ECA* optimization algorithm, presented in chapter 5, may also only use the training dataset. After the optimization and learning phase is completed, the generalization of the obtained classification pipeline is evaluated on the test dataset.

6.1.2 Results of the Optimization Algorithm

In the following a *single* optimization process of the *ECA-full* algorithm is conducted and evaluated to show specific but typical results. Note that more detailed analyses of this dataset are discussed in chapter 7 which provide, e.g., experiment repetitions to obtain statistical relevant results.

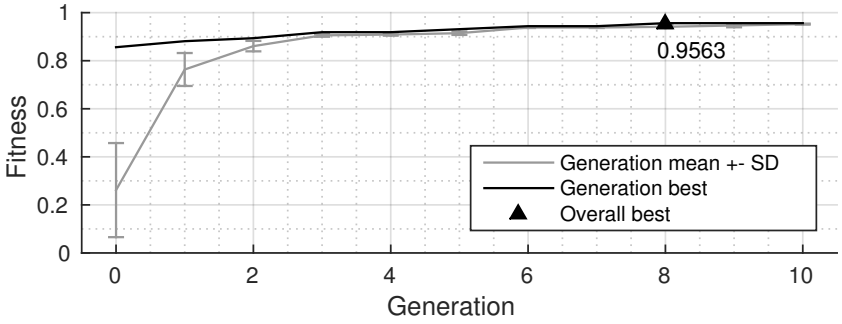


Figure 6.2: Fitness development during the optimization process of the *ECA-full* algorithm on the *coins* dataset.

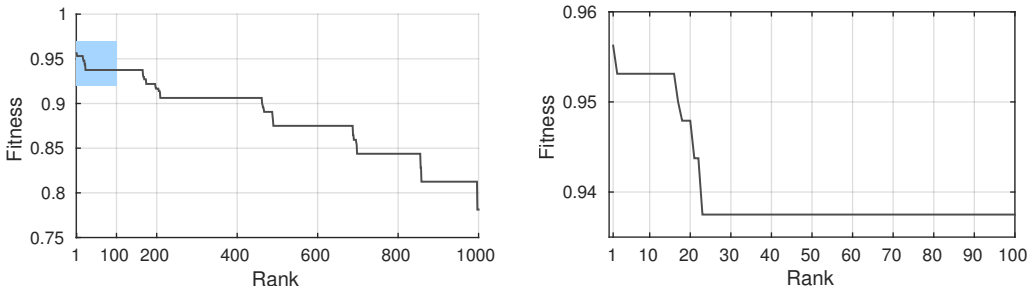
Rank	1	2	3
Fitness	0.9563	0.9531	0.9531
Feature subset	299 features	326 features	319 features
Feature preprocessing	Rescaling	Rescaling	Rescaling
Feature transform	LMNN, D_{Trans}=299, N_{Neigh}=3	LMNN, D_{Trans}=326, N_{Neigh}=1	LMNN, D_{Trans}=319, N_{Neigh}=3
Classifier	Gaussian SVM, c_{reg}=167.90, γ_{Gauss}=9.18·10^{-4}	Linear SVM, c_{reg}=394.55	Linear SVM, c_{reg}=56.51

Table 6.1: Exemplary top three configurations of the *ECA-full* algorithm for the *coins* dataset. The fitness value is equivalent to the corresponding cross-validation accuracy. The full configuration trajectory contains 2,345 configurations.

Optimization Trajectory

The proposed *ECA* optimization algorithm is based on Evolutionary Algorithms and therefore, it improves the solutions during multiple generations. Figure 6.2 shows the fitness development during the optimization process. It becomes apparent that the maximum fitness value of 0.9563 is reached after eight generations of continuous improvements. The average population fitness is steadily increasing while the spread of the population fitness values is decreasing over time. After ten generations this particular optimization process terminates with a total duration of 95.83 minutes.

After the optimization process the optimization trajectory is obtained with a ranked configuration list (see section 5.5.5), which usually contains several hundreds up to thousands of configurations. Table 6.1 presents the best three configurations and the corresponding fitness values of the *ECA-full* algorithm. The first observation is that the best three configurations chose the same or very similar algorithm combinations, namely rescaling for the feature



(a) Top 1,000 configurations (b) Top 100 configurations

Figure 6.3: Ranked fitness distribution of the optimization trajectory from the *ECA-full* algorithm on the *coins* dataset. Figure (a) shows the best 1,000 fitness values whereas figure (b) zooms into the best 100 fitness values from the shaded rectangle area. The discrete steps in the distribution function result from the relatively small number of training samples that lead to discrete cross-validation accuracy values.

preprocessing, LMNN for the feature transform and the SVM for the classifier. However, the selected features are different and the hyperparameters differ notably.

The overall best configuration in itself is not the only interesting solution because it can be randomly picked and therefore it might be very "unusual". Figure 6.3 reveals the fitness distribution of the best 1,000 configurations. It becomes obvious that multiple well-performing solutions are found during the optimization. In this example the performance decrease in the best 25 configurations is less than two percentage points of cross-validation accuracy. The precise fitness distribution is expected to be different for each learning problem and optimization run. However, observations and experiments have shown that usually around 50 configurations[1] can be considered as well performing and, as a consequence, potentially useful. A deeper analysis of these "top" solutions is promising and is therefore discussed in the next sections.

6.2 Graphical Solution Analysis

The optimization algorithm provides a ranked list of configurations that can be exported and analyzed in tables, as shown in table 6.1. However, the analysis of 50 or more configurations using a table is not effective. Suitable visualization techniques have the potential to facilitate a fast understanding of the configuration distribution to obtain insights into the classification problem. The following aspects are relevant to the problem understanding:

[1]The number of configurations that are considered in the extended optimization analyses is a metaparameter of the AROMS-Framework (see appendix E).

- Which method or set of methods performs best?
- Which features are the most relevant?
- Which connections exist between features and methods?

The *multi-configuration graph* is introduced in the next section to provide answers to these questions.

6.2.1 Multi-Configuration Graph

Every configuration can be interpreted as a connection of the selected features and the chosen algorithms as these components are "working" together in the classification pipeline. These connections can be modeled as a graph structure in which features and algorithms are vertices (or nodes) and the connections between them are edges (or links). In order to obtain graphs with a reasonable level of complexity, a selection of relevant components of the pipeline configurations has to be made. The most relevant components are the features subset, the feature transform and the classifier. The feature preprocessing is considered as less relevant because it is not expected to drastically change the data structure itself, e.g., making a non-linear problem linear. The hyperparameters are also less relevant since they are specific for each algorithm.

The goal of the *multi-configuration graph* is a well understandable visualization of the set containing the best $N_{Configs}$ configurations from an optimization trajectory

$$\begin{aligned} bestConfigSet(N_{Configs}) = \{ & \\ & rankedConfigs(1), \\ & rankedConfigs(2), \ldots, \\ & rankedConfigs(N_{Configs})\} \,. \end{aligned} \tag{6.1}$$

Graph Construction from a Single Configuration

Every single configuration θ can be transformed into a graph, as depicted in figure 6.4. It can be represented by a tuple

$$configGraph(\theta) = (\mathfrak{V}, \mathfrak{E}, f_{\mathfrak{V}}, f_{\mathfrak{E}}, f_{\mathfrak{V},Label}) \tag{6.2}$$

in which $\mathfrak{V} = \{\mathfrak{v}_l\}$ is the set of vertices and $\mathfrak{E} = \{\{\mathfrak{v}_{l_1}, \mathfrak{v}_{l_2}\}\}$ with $\mathfrak{v}_{l_1}, \mathfrak{v}_{l_2} \in \mathfrak{V}$ is the set of undirected edges. All edges and vertices of the graph are weighted which is described with the two weighting functions $f_{\mathfrak{V}} : \mathfrak{V} \mapsto \mathbb{R}$ and $f_{\mathfrak{E}} : \mathfrak{E} \mapsto \mathbb{R}$. Furthermore, all vertices have a description label which is defined by the function $f_{\mathfrak{V},Label} : \mathfrak{V} \mapsto description\ string$.

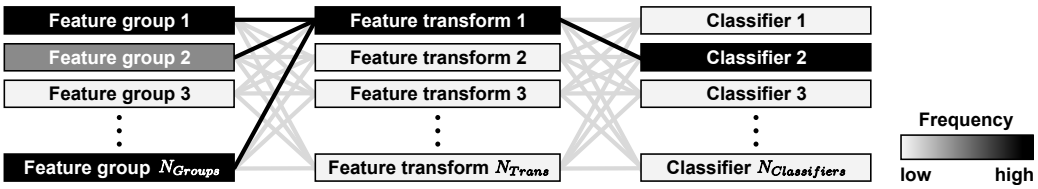


Figure 6.4: A graph representation of a single pipeline configuration that connects the selected feature groups, feature transforms and classifiers.

The weighting functions are used to describe the frequency of components and connections. The set of vertices is composed out of three sets

$$\mathfrak{V} = S_{FeatGroups} \cup S_{Trans} \cup S_{Classifiers} \,, \tag{6.3}$$

namely the set of feature groups $S_{FeatGroups}$, the portfolio set of feature transforms S_{Trans} and the portfolio set of classifiers $S_{Classifiers}$. The feature groups (see section 4.4) are used instead of the feature index set to handle multidimensional features as a single vertex, e.g., one feature group vertex for the LBP descriptor. The feature descriptions of the feature groups are used as a meaningful vertex label $f_{\mathfrak{V},Label}$ for the feature group vertices rather than the index number. The vertex labels for the feature transforms and classifiers are set according to the corresponding method name.

The structure of the graph is arranged according to the processing steps in the classification pipeline (see figure 4.2) so that the graph is easily understandable. There are two aspects that contribute to this idea. First, the vertices are arranged in three groups of columns so that the feature groups are on the left hand side, the feature transforms in the middle and the classifiers on the right hand side. Secondly, the edges connect the feature groups with the feature transforms as well as the feature transforms with the classifiers. More precisely, the set of edges is a fusion of two subsets

$$\mathfrak{E} = \mathfrak{E}_{Feat,Trans} \cup \mathfrak{E}_{Trans,Class} \tag{6.4}$$

with

$$\begin{aligned} \mathfrak{E}_{Feat,Trans} &= \{\{\mathfrak{v}_1, \mathfrak{v}_2\}\} \quad \text{with} \quad \mathfrak{v}_1 \in S_{FeatGroups} \wedge \mathfrak{v}_2 \in S_{Trans} \quad \text{and} \\ \mathfrak{E}_{Trans,Class} &= \{\{\mathfrak{v}_1, \mathfrak{v}_2\}\} \quad \text{with} \quad \mathfrak{v}_1 \in S_{Trans} \wedge \mathfrak{v}_2 \in S_{Classifiers} \,. \end{aligned} \tag{6.5}$$

The weighting function for the vertices is defined according to the occurrence of the specific components inside of the configuration

$$f_{\mathfrak{V}}(\mathfrak{v}) = \begin{cases} 1: & \mathfrak{v} \in S_{Trans} \wedge \mathfrak{v} = A_{Trans} \\ 1: & \mathfrak{v} \in S_{Classifiers} \wedge \mathfrak{v} = A_{Class} \\ w_{Group}(\mathfrak{v}): & \mathfrak{v} \in S_{FeatGroups} \\ 0: & else \end{cases} \tag{6.6}$$

in which A_{Trans} and A_{Class} denote the selected feature transform and classifier in θ, respectively. If a feature transform or classifier is selected, the weight of the corresponding vertex is simply set to the value one. The feature groups are more complex because they can be multidimensional: The weight of a feature group vertex is set to the fraction of features $w_{Group}(\mathfrak{v}) \in [0,1] \subset \mathbb{R}$ that are selected from that specific feature group $\mathfrak{v}$. The idea behind this weighting is a better visibility of the importance of a feature group without the need to visualize each sub feature in that group. Consider a feature group containing a SIFT feature with 128 dimensions while 50 of them have been selected in $S_{FeatSubSet}$. This leads to $w_{Group}(\mathfrak{v}) = 50/128 \approx 0.39$.

The weighting function for the edges takes the occurrences of the connections into account and is defined as

$$f_{\mathfrak{E}}\{\mathfrak{v}_1,\mathfrak{v}_2\} = \begin{cases} w_{Group}(\mathfrak{v}_1): & \{\mathfrak{v}_1,\mathfrak{v}_2\} \in \mathfrak{E}_{Feat,Trans} \ \wedge \mathfrak{v}_2 = A_{Trans} \\ 1: & \{\mathfrak{v}_1,\mathfrak{v}_2\} \in \mathfrak{E}_{Trans,Class} \ \wedge \mathfrak{v}_1 = A_{Trans} \wedge \mathfrak{v}_2 = A_{Class} \\ 0: & else \end{cases} . \tag{6.7}$$

If an edge connects a feature group with the selected feature transform A_{Trans}, the weight is set to the corresponding weight of the feature group vertex $w_{Group}(\mathfrak{v}_1)$. In case an edge connects the selected feature transform A_{Trans} with the selected classifier A_{Class}, the weight is set to the value one.

Fusion of Multiple Graphs

In order to obtain a configuration graph of multiple configurations, a graph fusion algorithm is proposed. Figure 6.5 visualizes the fusion process of "stacked" configuration graphs. The fusion algorithm uses a set of single configuration graphs $\{configGraph(\theta_l)\}$ with $1 \leq l \leq N_{Configs}$. The fused graph

$$configGraph_{Fusion}(\{configGraph(\theta_l)\}) = (\mathfrak{V}', \mathfrak{E}', f'_{\mathfrak{V}}, f'_{\mathfrak{E}}, f'_{\mathfrak{V},Label}) \tag{6.8}$$

has the same structure as a graph for a single configuration with the same set of vertices $\mathfrak{V}' = \mathfrak{V}$ and edges $\mathfrak{E}' = \mathfrak{E}$. Furthermore, the vertex labels are the same and $f'_{\mathfrak{V},Label} = f_{\mathfrak{V},Label}$. The fused weighting functions contain the common frequencies of the components – vertices and edges – across all $N_{Configs}$ configurations. More precisely, the fused functions are defined as

$$f'_{\mathfrak{V}}(\mathfrak{v}) = \sum_{l=1}^{N_{Configs}} f_{\mathfrak{V},l}(\mathfrak{v}) \ , \ \forall \mathfrak{v} \in \mathfrak{V} \tag{6.9}$$

and

$$f'_{\mathfrak{E}}(\{\mathfrak{v}_1,\mathfrak{v}_2\}) = \sum_{l=1}^{N_{Configs}} f_{\mathfrak{E},l}(\{\mathfrak{v}_1,\mathfrak{v}_2\}) \ , \ \ \forall \{\mathfrak{v}_1,\mathfrak{v}_2\} \in \mathfrak{E} \tag{6.10}$$

in which $f_{\mathfrak{V},l}$ and $f_{\mathfrak{E},l}$ denote the weighting functions of the lth configuration graph.

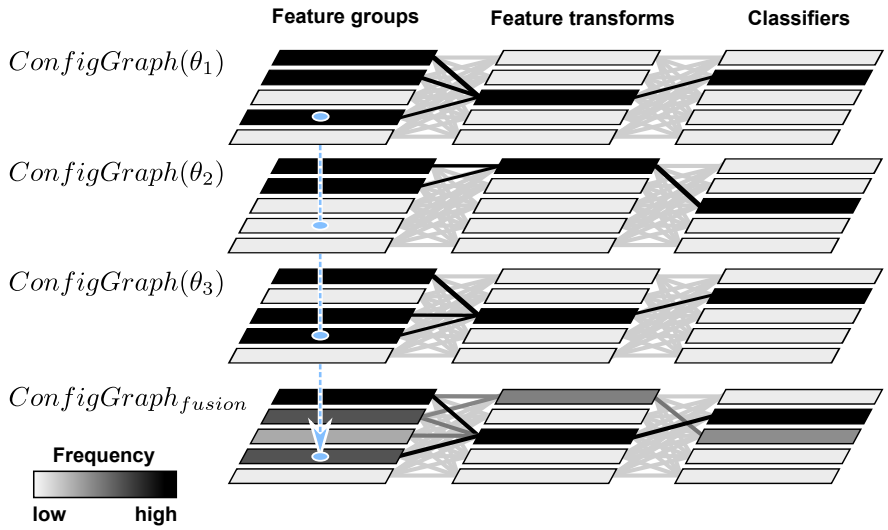


Figure 6.5: Visualization of the fusion process of multiple configuration graphs. The three-dimensional visualization shows the "stacking" of the configuration graphs to illustrate the weight fusion across all graphs at the same component (arrow).

Graph Optimization for Visualization

The resulting graph (see the bottom of figure 6.5) now contains all relevant information of the configuration set, however, it consists of too many components to use it directly for a well-understandable visualization. The total number of vertices in the graph is

$$|\mathfrak{V}| = |S_{FeatGroups}| + |S_{Trans}| + |S_{Classifiers}| \tag{6.11}$$

and the number of edges is

$$|\mathfrak{E}| = |S_{FeatGroups}| \cdot |S_{Trans}| + |S_{Trans}| \cdot |S_{Classifiers}| \,. \tag{6.12}$$

The dataset of the case study comprises 12 feature groups and the AROMS-Framework contains 31 feature transforms and 8 classifiers. This leads to a total number of $|\mathfrak{V}| = 12+31+8 = 51$ vertices and $|\mathfrak{E}| = 12 \cdot 31 + 31 \cdot 8 = 620$ edges in the graph, which will likely result in a graph that is difficult to understand. Therefore, multiple simplification and optimization steps are proposed to visualize the graph:

1. Vertices of the algorithms and edges with a weight of zero are removed from the graph as these components and links do not occur in any of the configurations and are thus not important for the classification task.
2. The position of the remaining vertices is ordered by their weight, so the top vertex in each of the three columns – feature groups, feature

transforms and classifiers – has the highest weight. This is a direct and naturally understandable sorting schema to quickly see the most important components.

3. The number of feature groups is limited to $N_{FeatGroupMax} = 16$ to preserve clarity. If more than this number of feature groups occur, a *rest vertex* is introduced and the remaining, least frequent and thus least important feature groups are fused into that vertex. The edges and their weights are fused accordingly as well.

4. The weights of the components are visualized with different shadings to further support the direct visibility of the component's importance. This is done for the vertices as well as for the edges. The shading is set to a linear interpolation between light gray to black in order to indicate the transition between the least frequent and the most frequent components. The shading values are normalized for each group of vertices $S_{FeatGroups}$, S_{Trans} and $S_{Classifiers}$ as well as for each group of edges $\mathfrak{E}_{Feat,Trans}$ and $\mathfrak{E}_{Trans,Class}$ so that in each group the full spectrum of shadings occurs.

5. The drawing order of the edges is chosen so that the ones with the lowest weight are drawn first. The goal is that the most important edges with the highest weight appear in the front and are not occluded by less important edges.

6. The fused configuration graph shows an "average" configuration. In order to still be able to identify the components that occur in the overall best configuration, the names of the corresponding vertices are denoted with an asterisk.

Case Study Results

Figure 6.6 presents the multi-configuration graph for the best $N_{Configs} = 50$ configurations for the case study using the *ECA-full* algorithm.

The feature group vertices show a smooth transition from important (dark shading) to less important (brighter shading) features. The area feature is the most important one, which is an expected result as the coins have a different size. The color features (mean and standard deviation of the hue channel) appear also very dark, which indicates their importance. This observation supports the expectation that the color is helpful to distinguish especially the 1-, 2- and 5-cent coins from the 10- and 20-cent ones. Interestingly, the mean gray value is irrelevant while its standard deviation is very important. Less important features are the low level pixel features, the local binary pattern variants (LBP) and the Hu moments. These features could potentially be removed to save computation time. Note that the number of feature groups is less than 16 here and thus no rest vertex is shown.

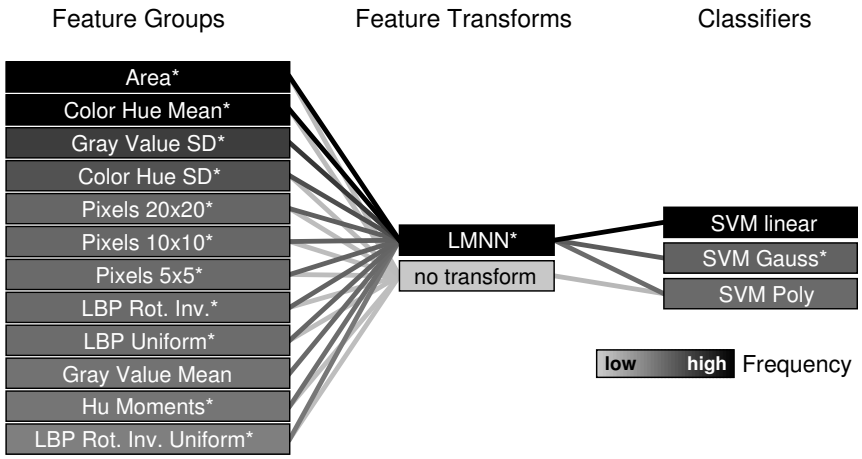


Figure 6.6: Multi-configuration graph for the best $N_{Configs} = 50$ configurations of the *ECA-full* algorithm on the *coins* dataset. The components that occur in the overall best solution are marked with an asterisk (*).

The analysis of the selected algorithms reveals that the diversity of algorithms that perform well on this dataset is rather small. The black shading indicates that the LMNN feature transform and the linear SVM classifier are selected in the clear majority of the solutions. This indicates that the LMNN transform generates features that allow a linear class separation in most cases. However, the overall best solution is the SVM with a Gaussian kernel, which is denoted with the asterisk. A minority of the configurations select no feature transform that works best with an SVM with a polynomial kernel. This indicates that the non-transformed classification problem is not well linearly separable.

6.3 Multi-Pipeline Classifier

The most obvious approach to use the proposed classification pipeline for actual classification tasks is to simply pick the fittest configuration after the optimization process and set up the classification pipeline with it. However, even the best classifier is likely not perfect and still causes errors for unseen instances, e.g., due to overfitting to the training dataset. A simple idea to improve the generalization performance is a *multi-pipeline classifier* (*MPC*) that makes use of the best configurations in the optimization trajectory. The combined decisions of a set or *ensemble* of multiple, independent classification pipelines for the same learning task are expected to be less prone to misclassifications. Figure 6.7 depicts this principle and shows that a fusion of multiple, simple and partly suboptimal classifiers can lead to a better decision boundary.

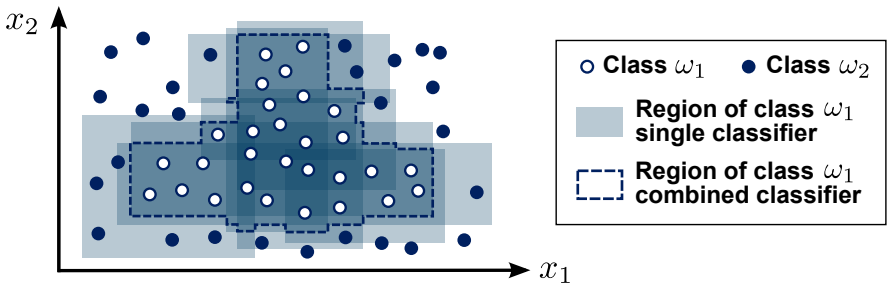


Figure 6.7: Principle of the combination of classifiers to improve the generalization in a two-class scenario. The shaded rectangular areas denote the class regions for class ω_1 that are predicted by multiple, simple classifiers. The combination of the single classifiers using majority voting results in a more precise class area denoted with darker shading. The complex decision boundary is shown with a dashed line.

6.3.1 Overview of Classifier Combination Approaches

The idea of combining multiple classifiers to improve the overall performance is well known in the field of machine learning for decades. There are numerous variants of this approach and several names exist, e.g., ensemble methods, classifier combination or fusion, mixture of experts or classifier committees [Polikar, 2006]. There are two main aspects of classifier combination methods: First, a *robust decision fusion* of multiple classifiers is required to even out single prediction errors. Secondly, the *diversity* of the involved classifiers needs to be reasonable. The more similar the models of the classifiers are, the less performance improvement can be expected of a combination of them. Consider a classifier combination consisting of, e.g., ten times exactly the same classifier with the same predictions. The combined result does obviously not lead to any benefit compared to the single classifier. It is worth noting that classifiers with the same cross-validation performance do not necessarily need to be completely equal. A different classifier model with, e.g., other hyperparameter values can predict different instance combinations of correctly and falsely classified vectors that lead to the same cross-validation accuracy value. However, the actual generalization performance can still be different and therefore, the fusion of multiple classifiers with the same cross-validation performance can potentially be useful.

This section only provides a rough overview of the most popular variants of classifier combination approaches. For a deeper insight, the reader is referred to, e.g., [Rokach, 2009], [Ranawana and Palade, 2006] or [Zhou, 2012]. For the sake of simplicity, the classifier combination methods are subdivided into two categories, which are described in the following.

Weighting Methods

The easiest way to combine multiple classifiers is the use of weighting methods. A set of classifiers is used to generate predictions for each input instance independently. The set of predictions is then combined with fusion methods such as

- *majority voting* to pick the most frequent class label,
- *performance weighting* that incorporates the accuracy performance estimation on the training dataset into the decision,
- *class probability weighting* which uses the conditional probability vector that a given instance belongs to a certain class or
- *complex fusion functions* using, e.g., machine learning approaches such as regression or classifiers to select the label based on the decisions and confidence values of the single classifiers.

Meta Combination Methods

Meta combination methods do not only affect the combination of the predictions but also other aspects such as the subdivision of the training datasets for different classifiers. Furthermore, most approaches combine so-called *weak classifiers* or *weak learners* that can be trained quickly, but only perform slightly better than guessing when they are solely used. The most popular meta combination methods are the following:

- *Bagging* is an abbreviation for "bootstrap aggregating" and was introduced by [Breiman, 1996]. For each classifier a "new" training dataset called *bootstrap sample* is randomly sampled with replacement[2] from the base training dataset. The final decision is realized by applying weighting methods to the predictions of the single classifiers. A popular example of this principle is the random forest classifier (see section 2.1.2).
- The principle of *Boosting* was introduced by [Schapire, 1990] and consists of multiple weak learners. The training process subsequently adds new classifiers that "focus" on the training samples which have been misclassified so far by increasing their weight.
- *Stacking* was proposed by [Wolpert, 1992] and is very similar to meta-learning (see section 3.1.1). Several different classifiers are used to classify an instance. The combination is done by a meta regression model that determines the optimal weights for the combination of the classifiers' predictions for each instance.

[2]The same training instance from the base dataset can appear multiple times in the new dataset.

6.3.2 Combination of Classification Pipelines

The work of [Kuncheva and Jain, 2000] serves as motivation for the proposed multi-pipeline classifier. They use genetic algorithms to select different but well performing feature subsets and train multiple classifiers using these feature subsets. The differences in these subsets are the crucial source of diversity along the classifiers. Finally, the predictions of the classifiers are combined and it is shown that the performance improves significantly.

The multi-pipeline classifier (*MPC*) extension of the AROMS-Framework exploits the best configurations of the optimization trajectory to set up multiple classification pipelines. The aspect of diversity along the classification pipelines is covered with the expected variation of the corresponding configurations regarding the feature subset, algorithm selections and hyperparameter values. Figure 6.3 shows that the *ECA* optimization algorithm finds multiple solutions with the same fitness/cross-validation accuracy value. As already mentioned in section 6.3.1, these classification pipelines do not necessarily need to be equal. Typically, they vary with respect to, e.g., the feature subset or the hyperparameter values so that the amount of diversity is reasonable. Therefore, it is expected that the degree of diversity in the *MPC* is higher than in the work of [Kuncheva and Jain, 2000].

After the training process the classification pipelines classify each instance independently and a fusion function is used to combine the predictions of each pipeline. The principle of the *MPC* is depicted in figure 6.8. The fusion function uses a majority voting schema, which is also applied in [Kuncheva and Jain, 2000]. The majority voting introduces the aspect of robustness to a certain amount of outliers within the predictions.

There is one noteworthy difference of the *MPC* compared to many popular approaches with similar goals. A single classification pipeline in the *MPC* would normally not be considered as a *weak classifier* because it contains highly adapted and potentially complex algorithms. However, it is expected that especially highly non-linear classification pipelines will tend to overfit to the training dataset and therefore, the multi-pipeline classifier extension has the potential to increase the generalization performance.

Note that a similar ensemble approach is also applied in a very recent version of the *auto-sklearn* framework [Feurer et al., 2015a], which uses Bayesian optimization to adapt classifiers (see section 3.2.5). The authors propose the use of the best solutions, which have been found during the optimization, to form a classifier ensemble. However, the idea and the results of the multi-pipeline classifier were first published several months earlier in [Bürger and Pauli, 2015c].

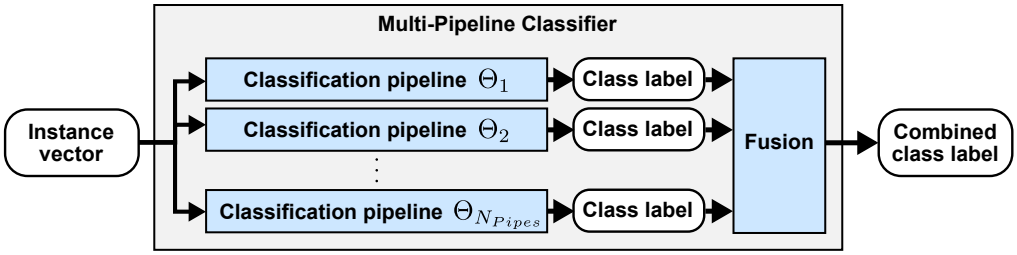


Figure 6.8: Processing of an instance using the proposed multi-pipeline classifier consisting of N_{Pipes} classification pipelines.

Training of a Multi-Pipeline Classifier

Before a multi-pipeline classifier can be used, an *ECA* optimization process has to be performed. The optimization trajectory – in form of the ranked configuration list $rankedConfigs(l_{rank})$ and the corresponding fitness values $rankedConfigsFit(l_{rank})$ – is required (see section 5.5.5). A multi-pipeline classifier is a list of $N_{Pipes} \in \mathbb{N}$ pipelines

$$MPList(T_{Train}, N_{Pipes}, rankedConfigs, rankedConfigsFit) = \\ [\Theta_1, \Theta_2, \ldots, \Theta_{N_{Pipes}}]\,. \tag{6.13}$$

The optimization trajectory just contains the configurations with the hyperparameters but not ready-to-use classification pipelines. Even though the pipelines have been trained during the holistic cross-validation method, they – or more precisely, the internal model parameters – are not permanently stored due to memory efficiency reasons[3]. Therefore, the l_{rank}th classification pipeline has to be trained again by

$$\Theta_{l_{rank}} = f_{TrainPipeline}(rankedConfigs(l_{rank}), T_{Train}) \tag{6.14}$$

with $1 \leq l_{rank} \leq N_{Pipes}$ using the training dataset T_{Train} and the l_{rank}th best configuration from the optimization trajectory.

Classification using a Multi-Pipeline Classifier

Once the multi-pipeline classifier is trained, it can be used to classify a new instance vector $\mathbf{x}$. This is done by processing the vector with all trained classification pipelines in the $MPList(\cdot)$ to obtain the predictions

$$y_{l_{rank}} = f_{ProcessPipeline}(\Theta_{l_{rank}}, \mathbf{x}) \tag{6.15}$$

with $1 \leq l_{rank} \leq N_{Pipes}$. The processing of each pipeline is independent from the others and can easily be done in parallel. The result is a list of predictions

[3]This aspect is also discussed in the outlook of this work in section 8.3.1.

$$predictionList(MPList(\cdot), \mathbf{x}) = [y_1, y_2, \ldots, y_{N_{Pipes}}]\,. \tag{6.16}$$

The fusion function combines the predictions to a scalar output using the majority voting method. This method chooses the most frequent class label along all predictions and is therefore robust to outliers. The majority voting is implemented as

$$y_{combined} = \underset{\omega_k \in S_{Classes}}{\arg\max} \sum_{l_{rank}=1}^{N_{Pipes}} sameLabel(y_{l_{Rank}}, \omega_k) \tag{6.17}$$

with the help of a class label comparison function

$$sameLabel(y, \omega) = \begin{cases} 1: & y = \omega \\ 0: & else \end{cases}. \tag{6.18}$$

The number of votes for each class is counted and the class with the highest number is returned. The special case of $N_{Pipes} = 2$ pipelines is problematic because the event of different predictions always ends in a tie. Therefore, this case is treated with $y_{combined} = y_1$ so that in case of two different predictions the one of the best pipeline is returned.

6.3.3 Selection of the Optimal Number of Pipelines

The number of classification pipelines $N_{Pipes} \in \mathbb{N}$ is a crucial metaparameter of the multi-pipeline classifier. On the one hand, a large number of pipelines is useful to enhance the diversity along the configurations. On the other hand, the fitness of the configurations is decreasing, the higher the rank l_{Rank} is chosen (see figure 6.3), and so the performance of the single pipelines becomes worse. Furthermore, the more pipelines are chosen, the higher the computational costs become to predict class labels. A trade-off between generalization and processing speed needs to be made. In the following, two strategies to determine the metaparameter N_{Pipes} are proposed.

Static Selection

The most straightforward strategy is the selection of a static number of pipelines, e.g., $N_{Pipes} = 20$. An advantage is its simplicity and the expected high diversity, if N_{Pipes} is chosen large enough. The fact that the actual fitness of the configurations is not considered at all is a clear disadvantage.

This strategy is denoted as $MPC^{Static}_{N_{Pipes}}$ because it uses the top-N_{Pipes} pipelines. As an example, MPC^{Static}_{20} uses the best 20 configurations.

Fitness-Dependent Selection

An extended selection strategy determines the number N_{Pipes} depending on the actual fitness of the configurations. A threshold $\Delta_{fit,max} \in \mathbb{R}$ is defined that limits the maximum fitness deviation of the configurations compared to the best one. The fitness difference of the configuration with the l_{Rank}th rank compared to the best configuration is calculated as

$$\Delta_{fit}(l_{Rank}) = rankedConfigsFit(1) - rankedConfigsFit(l_{rank}) \in \mathbb{R}\,. \tag{6.19}$$

The highest number of pipelines that fulfills this threshold criterion is determined using

$$N_{Pipes} = \underset{1 \leq l_{Rank} \leq N_{Pipes,max}}{\arg\max} \; l_{Rank} \cdot rankBelowThresh(l_{Rank}) \tag{6.20}$$

with the threshold criterion

$$rankBelowThresh(l_{Rank}) = \begin{cases} 1: & \Delta_{fit}(l_{Rank}) < \Delta_{fit,max} \\ 0: & else \end{cases}. \tag{6.21}$$

The maximum number of pipelines that should be considered is limited to $N_{Pipes,max} \in \mathbb{N}$ and the choice of $N_{Pipes,max} = 50$ keeps the computational effort at a reasonable amount. Furthermore, it is expected that configurations at higher[4] indices l_{Rank} only contain too poorly performing configurations that could even decrease the performance of the multi-pipeline classifier.

This variant of the multi-pipeline classifier is denoted as $MPC^{Fitness}_{\Delta_{fit,max}}$ and $\Delta_{fit,max} \in [0, 0.10] \subset \mathbb{R}$ is considered as a useful range of fitness thresholds. To illustrate an example, $MPC^{Fitness}_{0.02}$ uses all configurations with less than two percentage points of cross-validation accuracy decline compared to the best configuration.

6.3.4 Case Study Results

In the following, the results of the proposed multi-pipeline classifier on the case study dataset are evaluated. The goal is the improvement of the generalization performance on the test dataset. The generalization accuracy value of 0.9375 originating from the best single pipeline on the test dataset serves as a comparison, which can be found in figure 6.9 for $N_{Pipes} = 1$. This figure also shows the generalization accuracy development depending on the number of combined pipelines N_{Pipes}, which is equivalent to the $MPC^{Static}_{N_{Pipes}}$ strategy. An strong improvement of the generalization accuracy values of up to more than three percentage points can be noticed when multiple pipelines

[4]The ranked configuration list is sorted by the fitness values in a descending way so that higher indices or ranks contain configurations with a lower fitness.

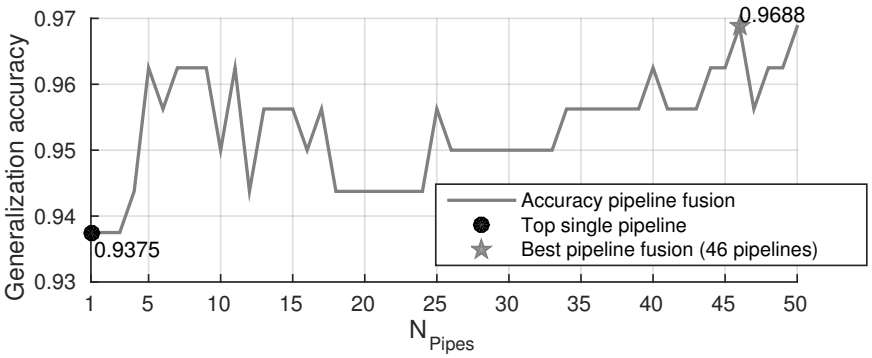


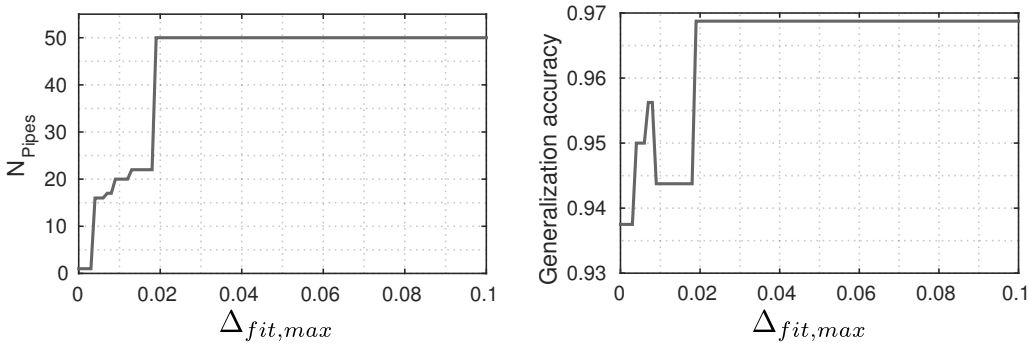
Figure 6.9: Generalization accuracy on the *coins* dataset of a multi-pipeline classifier depending on the number of pipelines.

are used. The best result is obtained for $N_{Pipes} = 46$ pipelines, but there are combinations with a smaller number of pipelines that perform only slightly worse, e.g., $N_{Pipes} = 5$ also leads to a generalization accuracy of more than 0.96. The resulting curve shows a high amount of variance, which indicates that the selection of a static number of pipelines will likely not lead to an optimal generalization performance for every dataset. Note that it would be unfair to determine the optimal value of N_{Pipes} with the help of the test dataset as the test dataset must not be used to tune any algorithm hyper- or metaparameter.

The results for the fitness-dependent selection of N_{Pipes} are shown in figure 6.10. As expected, the number of pipelines is quickly increasing with a rising fitness threshold value $\Delta_{fit,max}$ (see figure 6.10 (a)). Figure 6.10 (b) indicates that the generalization accuracy also rapidly increases to a value of more than 0.955 for $\Delta_{fit,max} = 0.007$, drops slightly afterwards and finally reaches a stable maximum value of 0.9688 for threshold values of $\Delta_{fit,max} > 0.02$. The fitness-dependent selection of the number of pipelines is more promising to achieve a good generalization because the resulting curve is far less "noisy" than the one for the static selection. However, the optimal fitness threshold needs to be evaluated on a wider range of datasets as also this metaparameter should not be tuned using the test dataset.

6.4 Discussion

This chapter presents two kinds of extensions for the AROMS-Framework that make use of the optimization trajectory, which is a "by-product" of the *ECA* optimization algorithm. The first extension is the multi-configuration graph that visualizes the distribution of the best configurations. The visual way of analyzing configurations is much more effective than studying tables with the configuration components. The proposed graph allows a quick and



(a) Number of pipelines

(b) Generalization accuracy

Figure 6.10: Impact of the fitness threshold for the multi-pipeline classifier on the number of pipelines and the generalization accuracy for the *coins* dataset.

intuitive overview of important features, algorithms and connections between them. The first results on the case study are promising, however, the usefulness needs to be shown on a larger variety of tasks, which is done in the next chapter.

The second extension is the multi-pipeline classifier that aims at improving the generalization performance. The best configurations of the optimization trajectory are used to set up multiple pipelines that classify each instance independently. The outcome is fused by majority voting to enhance the robustness against outlier predictions. The results on the case study dataset are promising and show a relatively large enhancement of the generalization accuracy. However, the choice of the optimal metaparameters needs to be evaluated on more datasets, which is done in the next chapter. Furthermore, the generalization improvement comes with the cost of higher computational costs. It depends on the application, whether a higher computational cost is tolerated for an improved generalization accuracy.

Chapter 7

Evaluation

The previous three chapters describe the AROMS-Framework which can be considered as a general purpose solution for the classification problem. In chapter 6 results of a single application of the AROMS-Framework on a case study dataset are already presented to motivate extended analyses using realistic data. However, the whole framework needs to be tested on multiple learning tasks in order to show its usefulness and also possible drawbacks of the approach. Thus, this chapter presents suitable evaluations of the AROMS-Framework and discusses the results.

The sections of this chapter are organized as follows. Section 7.1 describes and motivates the chosen evaluation approach. It covers the selection of datasets, the evaluation criteria and framework variants that are investigated. Furthermore, alternative methods for comparison reasons are listed. Section 7.2 presents extended results for the case study dataset, which is also used in the previous chapter. In section 7.3 the results for a real-world classification problem that occurs in the image-based measurement system for the cleanliness of steel are discussed, which is described in the introduction of this work. Section 7.4 evaluates the AROMS-Framework on multiple, publicly available classification datasets from different application fields. Comprehensive results of the AROMS-Framework across all datasets are presented in section 7.5. Finally, section 7.6 discusses the evaluation results.

7.1 Evaluation Approach

The proposed AROMS-Framework consists of three parts: the classification pipeline, the *ECA* optimization algorithm to adapt the pipeline configuration to a given learning task and the extended analyses that exploit the optimization process. Therefore, the evaluation approach pursues several aims. At first, it is necessary to determine if the AROMS-framework can improve the generalization performance for a wide range of classification tasks compared to standard state-of-the-art approaches. Secondly, it is important to find out

if the required optimization runtime is reasonable regarding the expected generalization improvements. Thirdly, the usefulness of the proposed extended optimization analyses needs to be proved on a wide range of tasks. This comprises the effectiveness of the multi-configuration graph as well as the generalization improvement of the multi-pipeline classifier.

There are four reasons why a suitable evaluation approach, which aims at achieving the aforementioned goals, is challenging:

- The application field of the AROMS-Framework is huge as it basically covers any kind of supervised classification task.
- The AROMS-Framework produces several kinds of output target metrics and data, e.g., the cross-validation accuracy, the generalization accuracy, the optimization runtime, the multi-configuration graph and the multi-pipeline classifier.
- The effects of multiple variants, design decisions and metaparameters of the AROMS-Framework on the target metrics need to be quantified in order to understand the internal functionality of the framework.
- The proposed *ECA* optimization algorithm of the AROMS-Framework is not deterministic because Evolutionary Algorithms rely on random processes.

Consequently, the proposed evaluation approach needs to consider multiple criteria on multiple datasets with statistical analyses of the repetitions of all experiments. The following subsections provide details about the evaluation approach.

7.1.1 Dataset Selection

The field of all possible classification tasks is literally infinitely large and therefore, a suitable selection has to be made. One of the main motivation for this thesis is the field of image-based object classification, which arises from the cleanliness measurement system for steel samples (see section 1.1). Image-based object recognition systems usually comprise two stages, namely the object detection inside of an image scene as well as the object classification. The focus of this work is the field of machine learning and classification. Therefore, the optimization of the detection and segmentation of objects inside of images is not considered in this work[1]. Furthermore, the particular challenge of the steel object classification task is the impact of the curse of dimensionality originating from the relatively low number of training samples in combination with the high feature space dimensionality.

[1]The simultaneous optimization of image processing and machine learning is discussed in the outlook of this work in section 8.3.3.

Consequently, two datasets with image-based classification tasks are analyzed in detail that share the aforementioned properties. The first one is the *coins* dataset of the case study (see section 6.1) and the second one is the steel object classification task, which will be denoted as *steel* dataset. The corresponding results can be found in section 7.2 and 7.3, respectively.

In order to show the universality of the AROMS-Framework, a larger variety of 11 classification tasks of publicly available databases is analyzed in section 7.4. These datasets cover different application fields such as medical diagnose systems, forensic applications and handwritten digits recognition. The analyses on these datasets will be limited to the most important basic criteria only. Otherwise immense amounts of statistics would have to be discussed that would only contribute few additional knowledge.

Note that the AROMS-Framework is designed for the classification of one-dimensional data and that the input basically consists of one or multiple vectors (see section 4.4). Image data is inherently two-dimensional and usually, suitable features need to be derived from the image textures to obtain one-dimensional feature vectors. However, the automatic development of image features is also not directly[2] considered within this work and the feature set has to be provided by the developer of the system. This is the reason why most of the datasets in this evaluation concept do not originate from image-based applications. Furthermore, popular image categorization benchmarks such as *Caltech101*[3] [Fei-Fei et al., 2007] are not considered in the evaluation as they provide pure image data. However, the AROMS-Framework could be used as a part of the solution for these image-based applications.

7.1.2 Evaluation Criteria

There is a variety of output metrics and data from each application of the AROMS-Framework on a classification task, which is described in the following.

Central Performance Criteria

The most important performance metrics are the following *central performance criteria*:

- The best *cross-validation accuracy* during the training process is equivalent to the fitness value of the best individual. This metric is helpful

[2]The feature transforms in the classification pipeline (see section 4.5.3) are able to generate a new feature representation that could be better suitable for the classifier than the input features. However, it is not expected that pure pixel data is sufficient to learn better features that outperform state-of-the-art feature descriptors for any dataset – especially when only few training samples are available.

[3]The *Caltech101* dataset is a database for the image categorization problem with 101 object and scene categories. Each category contains around 50 color images on average.

to measure the adaptation of the AROMS-Framework to the learning task defined by the training dataset T_{Train}.

- The *generalization accuracy* of the overall best configuration on the test dataset T_{Test} is of central interest as a maximum generalization performance is desired. Note that the cross-validation accuracy is an estimation of the generalization accuracy (see section 5.1). In the ideal case these two values are equal, however, in real applications significant deviations between them can occur.
- The *optimization runtime* is of interest as well since a significant longer optimization runtime is usually not desired for only a marginal accuracy improvement. This metric is important when the *Pareto principle*[4] is considered.
- The *classification time* or classification speed per instance is important for applications which require fast classifiers. The classification time is measured with the test dataset which is processed by the classification pipeline resulting from the best configuration.

Criteria of the Extended Analyses

The following criteria are used to evaluate the proposed extended analyses of the optimization process (see chapter 6):

- Typical examples of the *multi-configuration graph* are presented and discussed. The full variety of all solutions, however, cannot be listed within this work.
- The generalization accuracy values of the *multi-pipeline classifier* are presented depending on the number of pipelines and the fitness threshold. The performance is compared to the generalization of the single classification pipeline.

7.1.3 Comprehension of the Framework Functionality

The AROMS-Framework is a complex system containing numerous components that interact with each other. First, there is the central classification pipeline consisting of four pipeline elements with different processing steps and algorithm portfolios. And secondly, the proposed *ECA* optimization algorithm is controlled by numerous metaparameters and design decisions. The overall interplay of these components is expected to be highly complex

[4]The Pareto principle, also known as the "80–20 rule", states that roughly 80% of the desired effect, e.g., the accuracy, are caused by only about 20% of the effort, e.g., the optimization runtime [Pareto, 1964]. It is often the case that the missing steps to perfection would require an unreasonable amount of effort.

and therefore, a deeper understanding of it is necessary. This is achieved by the evaluation of different variants and metaparameters of the AROMS-Framework.

Impact of Variants of the Pipeline Components

The optimization chapter presents different variants of the *ECA* optimization algorithm (see section 5.6). These variants differ with respect to the classification pipeline components that are optimized with the goal to quantify their impact. The standard variant is the *ECA-full* algorithm with the highest amount of adaptability. The other five variants are summarized as follows: The *ECA-noFeatSel* variant is used for analyzing the aspect of feature selection, the *ECA-noPreProc* variant for the preprocessing, the *ECA-noTrans* variant for the feature transform, the *ECA-simpleClassifier* variant for the classifier and the *ECA-defaultHyper* variant for the hyperparameter tuning. These variants are directly compared to the *ECA-full* variant with respect to the central performance criteria.

Impact of Selected Design Decisions of the ECA Algorithm

It can be expected that the extended Evolution Strategies that are used by the *ECA* algorithm also have a large impact on the results. It is desirable to investigate selected design principles and metaparameters of the optimization algorithm itself to understand the resulting effects:

- The impact of the *improvement of the initial population* using knowledge about the feature relevance from random forests (see section 5.5.4) is compared to a completely random initial population.
- The effect of the *holistic cross-validation* method (see section 5.1.2) is compared to a cross-validation approach that only considers the generalization of the classifier itself. In this version the training dataset T_{Train} is not subdivided into training and validation sets until it is passed to the classifier pipeline element. The feature transform is trained with the full training dataset so that it never needs to process unseen data in this *classifier-only cross-validation* variant.
- The role of the *early discarding* system (see section 5.1.3) within the holistic cross-validation is of special interest. On the one hand, the expected processing speedup needs to be quantified. On the other hand, it needs to be checked if negative effects occur due to the replacement of the average cross-validation with an estimation[5] of it.

[5]The early discarding system can prematurely stop the cross-validation process and thus it may influence the accuracy estimation. Performance underestimations become more likely due to the aggressive discarding criteria.

Of course, the *ECA* optimization algorithm is controlled by more metaparameters that could potentially be modified, e.g., the evolutionary metaparameters such as the initial population size μ_{init} or the number of parents ρ (see section 5.5.3 and appendix E). However, these metaparameters are empirically tuned and determined in preliminary studies [Bürger and Pauli, 2015b, Bürger and Pauli, 2016]. In order to keep the amount of studies reasonable within this work, the impact of changes regarding these metaparameters is not quantified.

7.1.4 Repetition Analysis and Statistical Tests

The problem of non-deterministic effects of the optimization algorithm is tackled with repetitions of all experiments. The objective function with its numerous local optima (see section 5.2) is expected to be the cause of inferior solutions that influence the variation of all target metrics. Therefore, not only the means but also the standard deviations have to be analyzed. The experiments with the AROMS-Framework are repeated ten times, which represents a compromise between computational effort and statistical relevance.

Furthermore, in order to argue that certain variants of the AROMS-Framework are superior compared to others, it is not sufficient to simply compare the means of the performance metrics. The risk of random effects explaining differences between the algorithms is high. Therefore, a suitable statistical test has to be conducted to prove that the results of algorithms significantly differ or not. The *Welch's unequal variances t-test*, or simply *Welch test*, is appropriate for this purpose, which is justified and explained in appendix F. A commonly used level of significance of $\alpha_{Welch} = 0.05$ is applied. As it is a pairwise test, all investigated variants are compared to the standard variant of the AROMS-Framework, namely the *ECA-full* optimization algorithm. If a significant difference – either positive or negative – is detected, the corresponding item is marked with an exclamation mark (!).

7.1.5 Baselines and Competing Approaches

The first step in the development of any machine learning system should be the evaluation of state-of-the-art standard methods. These so-called *baseline* methods reveal how much performance can be achieved with a minimum amount of experience in the field of machine learning. A new and complex classifier system would be futile, if it did not perform significantly better than such baseline methods.

Comparison with a Baseline SVM and Random Forest

The support vector machine (SVM) is very popular and performs well on a wide range of tasks. The *baseline SVM* is an SVM with a Gaussian kernel which allows a non-linear class separation. In order to achieve the best performance, the two hyperparameters c_{reg} and γ_{Gauss} are tuned based on basic grid search with a grid sampling density of three values per hyperparameter (see appendix D). The hyperparameter combination with the highest cross-validation accuracy on the training dataset is chosen. Besides the hyperparameter tuning, no other methods, e.g., feature selection or preprocessing, are involved for the baseline classifier.

Furthermore, the popular concept of random forests is also considered as it has a built-in feature selection and usually performs well on complex and high-dimensional datasets. Therefore, the random forest is the second baseline classifier, which is denoted as *Baseline RF*. Its hyperparameters are tuned in the same way as the baseline SVM classifier.

Comparison with Auto-WEKA

The machine learning community has developed numerous frameworks that automatically optimize the performance of machine learning systems (see section 3.2.5). The *Auto-WEKA* framework [Thornton et al., 2013] is a state-of-the-art optimization framework which optimizes features, classifiers and hyperparameters. Therefore, it is well suited to compare its generalization performance. Its standard metaparameters determine a fixed time budget for the optimization which is set to 24 hours. The long processing time is the reason why the compared experiments are only repeated five times instead of ten times. Version 0.5 of the *Auto-WEKA* framework is used, which does not take advantage of parallel processing on multiple processors while the AROMS-Framework uses six parallel processes (see section 7.1.6). The reason for this drawback of the *Auto-WEKA* framework is the difficulty to parallelize the Bayesian optimization approach. It might appear unfair to compare the AROMS-Framework to *Auto-WEKA* as the AROMS-Framework can process six times more solutions in the same time span. However, for *Auto-WEKA* a time budget of 24 hours is theoretically equivalent to $24/6 = 4$ hours of time budget for the AROMS-Framework. This fact has to be kept in mind when the optimization runtimes are compared.

7.1.6 Computing Environment

The computer and software system that is used to evaluate the AROMS-Framework has the following properties. A workstation computer with an *Intel Xeon* processor having six physical processing cores at 2.50 Gigahertz and 32 Gigabytes of memory (RAM) is used. As operating system the Linux

	Cross-validation accuracy	Generalization accuracy
ECA-full	**0.9350** ± 0.0154	**0.9513** ± 0.0128
Baseline SVM	0.1187 ± 0.0125 (!)	0.2812 ± 0.0198 (!)
Baseline RF	0.6231 ± 0.0281 (!)	0.7081 ± 0.0453 (!)
Auto-WEKA	(not comparable)	0.9300 ± 0.0143 (!)

Table 7.1: Comparison of cross-validation and generalization accuracy results for the *coins* dataset with mean $\pm 1\sigma$. Statistically significant deviations compared to the proposed *ECA-full* algorithm are denoted with an exclamation mark. Note that the *Auto-WEKA* framework uses a different optimization metric which is not directly comparable to the cross-validation accuracy in the AROMS-Framework.

distribution *Ubuntu* in version 14.04 is used. The AROMS-Framework is implemented in Matlab version 2014b. The *Matlab Parallel Computing Toolbox* is used to speed up the *ECA* optimization algorithm. The fitness evaluation of the individuals, which is the computationally most expensive part of the *ECA* algorithm, runs in parallel using six processes.

7.2 Case Study: Classification of Coins

The first evaluations are conducted on the *coins* dataset, which is introduced in the previous chapter. The basic goal of this task is to distinguish five classes of euro coins, which are depicted in figure 6.1. A set of image-based features is used that is precisely described in section 6.1. The previous chapter already analyzes the results of a *single* application of the AROMS-Framework on this dataset. This section presents the results of all relevant framework variants with repetitions of the experiments. Note that the results of this particular section are already published in [Bürger and Pauli, 2016].

7.2.1 Central Performance Criteria

This section presents the central performance criteria of the *ECA-full* optimization algorithm for the *coins* dataset.

Accuracy Values

Table 7.1 shows the cross-validation and generalization accuracy values compared to the two baseline methods and the *Auto-WEKA* framework. The proposed *ECA-full* algorithm performs significantly better compared to all other methods with respect to the cross-validation and generalization accuracy. The two baseline methods perform poorly on this dataset, which

	Best fitness	Best generalization
Cross-validation accuracy	**0.9563**	0.9250
Generalization accuracy	0.9375	**0.9625**
Feature subset	299 features	333 features
Feature preprocessing	Rescaling	Rescaling
Feature transform	LMNN, D_{Trans}=299, N_{Neigh}=3	LMNN, D_{Trans}=333, N_{Neigh}=20
Classifier	Gaussian SVM, c_{reg}=167.90, γ_{Gauss}=$9.18 \cdot 10^{-4}$	Linear SVM, c_{reg}=165.58

Table 7.2: Best configurations of the *ECA-full* algorithm for the *coins* dataset with the highest fitness and generalization accuracy across the repeated experiments.

indicates that feature selection or preprocessing steps are particularly useful. Especially interesting is that the baseline SVM performs very poorly, even though this classifier usually works well on a wide range of tasks. The *Auto-WEKA* framework also finds fairly well performing solutions, however, compared to the *ECA-full* algorithm the average generalization performance is about two percentage points lower.

The standard deviation of the accuracy values achieved by the *ECA-full* algorithm is relatively low with values of 1.3 to 1.5%, which shows that the proposed method is robust and does not select inferior local optima frequently for this dataset. The results of the baseline random forest show significantly higher standard deviation values, e.g., more than ±4.5% of generalization accuracy. It can be expected that the inherently random processes inside of this classifier concept are responsible for this effect. Therefore, the random forest classifier is far less suitable for this dataset.

Best Configurations

The best configurations that have been found during the repeated experiments are listed in detail in table 7.2. It can be seen that the configuration with the highest cross-validation accuracy does not necessarily achieve the highest generalization accuracy. This shows that cross-validation is only an approximation for the actual generalization. The configurations, however, are relatively similar to each other. The same algorithms are involved, namely rescaling for the preprocessing, the feature transform LMNN and the SVM – with different kernels – for the classifier.

	Optimization runtime [min]	Classification time per instance [ms]
ECA-full	73.74 ± 20.28	**0.19** ± 0.05
Baseline SVM	**0.02** ± 0.01 (!)	0.41 ± 0.12 (!)
Baseline RF	0.48 ± 0.01 (!)	5.76 ± 2.52 (!)
Auto-WEKA	1,450.93 ± 28.33 (!)	(not available)

Table 7.3: Comparison of optimization runtime and classification time results for the *coins* dataset with mean $\pm 1\sigma$. Statistically significant deviations compared to the proposed *ECA-full* algorithm are denoted with an exclamation mark. Note that the classification times of the *Auto-WEKA* framework have not been logged.

Optimization Runtime

The optimization runtimes are presented in table 7.3. The *ECA-full* algorithm needs about 74 minutes on average to terminate the optimization process. At first glance these runtimes seem to be surprisingly fast as the complexity analysis of the configuration adaptation problem predicts a runtime of billions of years in case of naïve grid search (see section 5.2.2). However, the biggest contribution to the complexity arises from the feature selection problem, which is very efficiently handled by the Evolutionary Algorithm. In comparison to that the *Auto-WEKA* framework uses approximately 24 hours (=1,440 minutes) for each optimization, which is obviously significantly longer than the optimization runtime of the *ECA-full* algorithm. Of course, the runtimes of the *ECA-full* algorithm and *Auto-WEKA* are substantially longer than any baseline method that involves only a single classifier.

Classification Time

The classification times of the resulting classifiers can be found in table 7.3. The times are measured by classifying the test dataset and the required time is divided by the number of instances to obtain the classification time in milliseconds per instance. The average classification time of the classification pipelines found by the *ECA-full* algorithm is significantly lower than the classification times of the baseline classifiers. This might be surprising because the baseline classifiers are expected to be relatively simple and fast. However, it can be observed that for this particular dataset the linear SVM classifier is selected by the *ECA-full* algorithm in almost all repetitions of the experiment (see also the multi-configuration graphs in figure 7.3). The linear kernel classifies instances much faster than the Gaussian kernel of the baseline SVM. The random forest is also slower because of the relatively large amount of decision trees that is used to classify the instances.

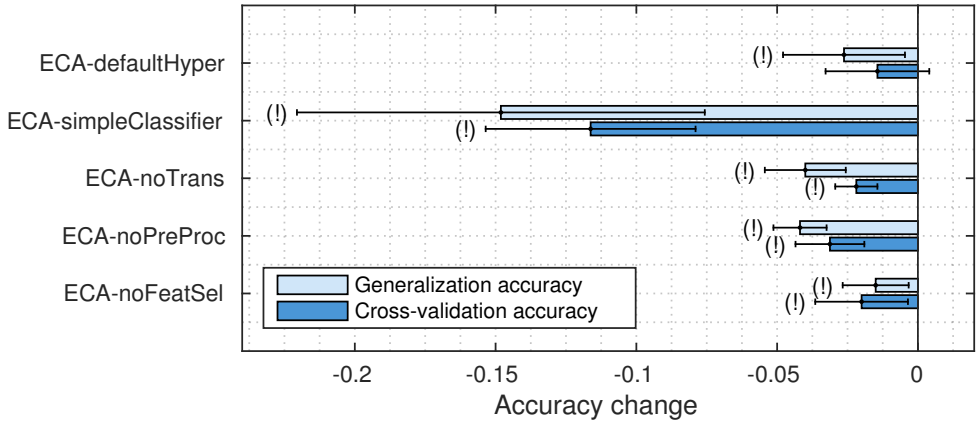


Figure 7.1: Impact of the *ECA* variants compared to the *ECA-full* algorithm on the cross-validation and generalization accuracy for the *coins* dataset. The rectangular bars show the average difference and the standard deviations are denoted through lines. Statistically significant deviations compared to the proposed *ECA-full* algorithm are denoted with an exclamation mark.

7.2.2 Impact of the Pipeline Components

This section analyzes the impact of the classification pipeline components, namely the feature selection, preprocessing, feature transform, classifier and hyperparameters on the accuracy and the optimization runtime. Therefore, five variants of the *ECA* algorithm (see section 5.6) are compared to the *ECA-full* variant.

Accuracy Values

The impact of the pipeline components on the cross-validation and generalization accuracy can be found in figure 7.1. All limited *ECA* variants perform worse than the *ECA-full* variant and besides the cross-validation accuracy of the *ECA-defaultHyper* variant, any differences are statistically significant. This means that all investigated components of the classification pipeline are important and contribute to an improvement of the accuracy for this particular dataset. Furthermore, a strong correlation between the average cross-validation accuracy during the training and the generalization accuracy can be observed for almost all variants of the *ECA* algorithm. This indicates the consistency of the generalization estimation by the holistic cross-validation method.

The impact of the classifier portfolio is especially large because the accuracy results of the simple naïve Bayes classifier – which is used in the *ECA-simpleClassifier* variant – are 10 to almost 15 percentage points worse on average. Compared to that, the accuracy decline of the other *ECA* variants is smaller and lies below five percentage points of accuracy. A first

approach to explain the results is that the classification pipeline is incapable to generate a suitable and "easy" feature representation. More complex classifier concepts such as the SVM are required to achieve a good performance. However, the fact that an SVM with a *linear* kernel has the overall best generalization accuracy (see table 7.2) indicates that the classification pipeline does achieve a reasonably simple feature representation. It is merely the case that the model of the naïve Bayes classifier is not suitable here.

Another interesting observation is that feature preprocessing seems to be equally important as feature transforms for this dataset. It could have been expected that the potential of feature transforms would have been larger due to their higher complexity. However, the success highly depends on the dataset and also on the amount of training samples.

The aspect of feature selection contributes rather less to the overall accuracy improvement. This indicates that the full feature set does not contain a too large fraction of noisy features so that the selection of a subset of features is not necessary to achieve a good performance. Furthermore, the aspect of hyperparameter tuning has only a small effect on the accuracy values for this dataset. It is presumably the case that the standard hyperparameters of the well performing classifiers do not require further tuning, which may happen by chance.

Optimization Runtime

Figure 7.2 shows the impact of the pipeline components on the optimization runtime. It becomes apparent that most variants lead to a faster average optimization runtime, except the *ECA-noFeatSel* variant. The missing feature selection results in a higher-dimensional feature space for all following algorithms that lead to a higher overall optimization runtime of more than 50 additional minutes on average. Especially the feature transform is responsible for the increase of the optimization runtime as the complexity of many algorithms depends on the number of dimensions. The feature transforms contribute the most to the optimization runtime. This is obvious in the *ECA-noTrans* variant that does not use feature transforms: The average optimization runtime is decreased by about 50 minutes compared to the *ECA-full* variant. The reasons for the high optimization runtimes of the feature transforms are two-fold. First, the computational models that need to be trained are rather complex and, secondly, the implementation in Matlab is not necessarily optimally efficient.

The *ECA-noPreProc* variant is more than 20 minutes faster than the *ECA-full* variant on average, which must be explained differently compared to the faster optimization runtimes of the *ECA-noTrans* variant: The computational models of feature preprocessing are simple and can be trained very quickly. However, one explanation is that more computationally expensive

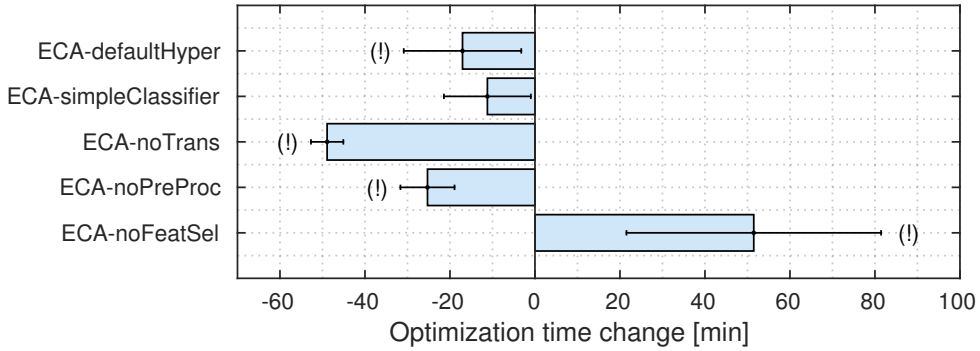


Figure 7.2: Impact of the proposed *ECA* variants compared to the *ECA-full* algorithm on the optimization runtime for the *coins* dataset. The rectangular bars show the average difference and the standard deviations are denoted through lines. Statistically significant deviations compared to the proposed *ECA-full* algorithm are denoted with an exclamation mark.

	Cross-validation accuracy	Generalization accuracy
ECA-full	0.9350 ± 0.0154	**0.9513** ± 0.0128
– random initial population	0.9119 ± 0.0217 (!)	0.9250 ± 0.0253 (!)
– classifier-only cross-validation	**1.0000** ± 0.0 (!)	0.2869 ± 0.0221 (!)
– no early discarding system	0.9381 ± 0.0237	0.9331 ± 0.0267

Table 7.4: Impact of selected design decisions of the *ECA* optimization algorithm on the cross-validation and generalization accuracy for the *coins* dataset with mean $\pm 1\sigma$. Statistically significant deviations compared to the proposed *ECA-full* algorithm are denoted with an exclamation mark.

feature transforms and classifiers perform better on the preprocessed data and are thus selected more often during the optimization process. A second explanation is the larger search space when preprocessing is considered.

7.2.3 Impact of Selected Design Decisions of the ECA Algorithm

This section describes the role of three important design decisions of the *ECA* optimization algorithm, namely the improvement of the initial population, the holistic cross-validation and the early discarding system. The accuracy values can be found in table 7.4 and the corresponding optimization runtimes in table 7.5. The results are discussed in the following.

Improvement of the Initial Population

The average cross-validation and generalization accuracy drops slightly about two percentage points, if a completely random initial population is used in-

	Optimization runtime [min]
ECA-full	73.74 ± 20.28
– random initial population	**59.30** ± 14.95
– classifier-only cross-validation	59.95 ± 9.16
– no early discarding system	276.15 ± 59.64 (!)

Table 7.5: Impact of optimization metaparameters on the optimization runtime for the *coins* dataset with mean $\pm 1\sigma$. Statistically significant deviations compared to the proposed *ECA-full* algorithm are denoted with an exclamation mark.

stead of the population with improved feature subsets (see section 5.5.4). The effect is rather small but still significant. It is interesting that the average optimization runtime drops marginally but not significantly. It is expected that the optimization process runs longer because the fitness of the initial solutions is worse with completely random individuals. It might be the case that the *ECA* optimization algorithm tends to prematurely terminate in local optima and thus requires less optimization runtime.

Holistic Cross-Validation

The effect of holistic cross-validation on the accuracy values is tremendous. The classifier-only cross-validation achieves an average cross-validation accuracy value of 1.0 – which would be perfect. However, the average generalization accuracy of 0.2869 is extremely bad so that the corresponding pipeline configurations are basically useless. The reasons for this effect are obvious: supervised feature transforms such as LDA, NCA and MCML can "cheat" when the classifier-only cross-validation is used because they calculate a feature representation that is basically the ground truth label distribution. To give an example, instances of class ω_1 could be projected to the value 1 and instances of class ω_2 to the value 2 and so on. A very simple kNN classifier achieves perfect results as the feature value can be directly mapped to the class label. However, this "magic" mapping of instances to a feature value according to their true class label is likely not working for unseen samples.

Furthermore, the average optimization runtime drops slightly but not significantly, if the classifier-only cross-validation is used. A possible explanation is the additional computational effort for holistic cross-validation, which turns out to be moderate for the *coins* dataset.

Early Discarding System

If the early discarding system is switched off, the average optimization runtimes increase tremendously to about 4.5 hours, which is around 3.7 times

slower. The average accuracy values are not affected in a negative way, at least not significantly.

7.2.4 Extended Optimization Analyses

This section presents the results of the multi-configuration graphs and the multi-pipeline classifier.

Multi-Configuration Graphs

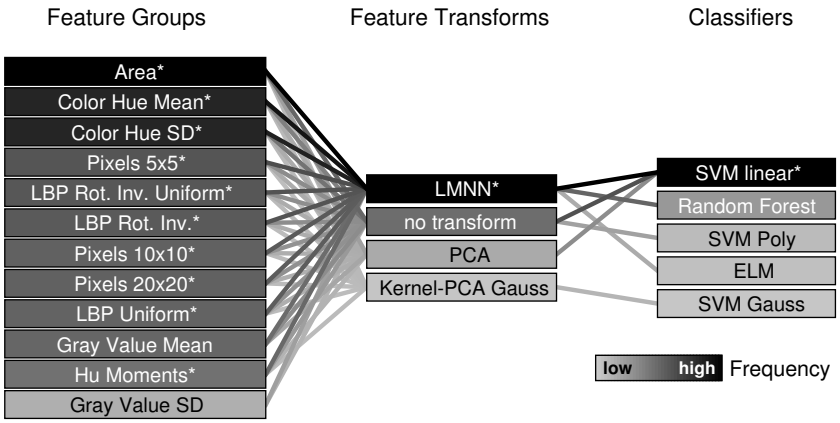
Figure 7.3 shows the multi-configuration graphs from two different and independent repetitions of the *ECA-full* algorithm on the *coins* dataset. First of all, the best and dominating configurations found by both optimization runs are similar, which can be quickly recognized due to the dark shading. In both cases, the feature transform LMNN and the linear SVM perform best. The feature set differs slightly, but the three most important features, namely the area and color features, appear even in the same order. The most obvious difference between the two graphs is the different amount of diversity of algorithms that are selected within the best 50 configurations. The graph in figure 7.3 (a) shows a large diversity of feature transforms and classifiers while the graph in figure 7.3 (b) indicates that in this run most of the best 50 solutions are relatively similar. The differences can be explained by looking at the different optimization trajectories: If many similar solutions appear under the best configurations, significant fine-tuning is performed during the end phase of the optimization. If the algorithm diversity is larger, it is likely the case that a good solution is found directly, which cannot be improved much more.

Multi-Pipeline Classifier

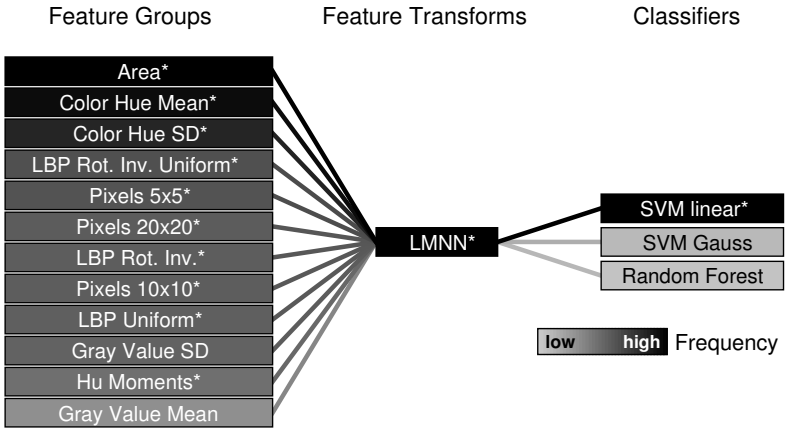
Figure 7.4 depicts the generalization accuracy values of the multi-pipeline classifier for the *coins* dataset depending on its main metaparameters.

The results for the static selection of the number of pipelines are shown in figure 7.4 (a). The generalization accuracy steadily increases in the range of $5 \leq N_{Pipes} \leq 15$ pipelines. After that, the average accuracy values increase only very slowly. The first statistically significant improvements compared to a single classification pipeline appear for $N_{Pipes} = 25$ pipelines, however, these are isolated effects. The generalization accuracy values reach a stable level of significant improvement of about 1.5 percentage points for $N_{Pipes} \geq 35$ pipelines compared to the single pipeline ($N_{Pipes} = 1$). It can be concluded that $N_{Pipes} \geq 35$ is recommended for this dataset.

Figure 7.4 (b) shows the results for the fitness-dependent selection of the number of pipelines. The general developing of the accuracy values is similar to the static selection but the curve shows less variance and thus it appears



(a) Large diversity of components

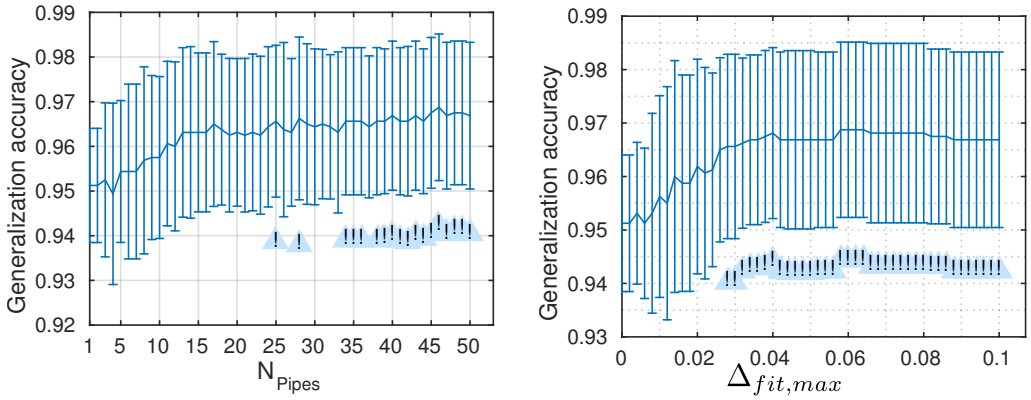


(b) Small diversity of components

Figure 7.3: Variations of the proposed multi-configuration graph of the *ECA-full* algorithm for the *coins* dataset with $N_{Configs} = 50$. The graphs are generated from two independent repetitions of the experiment.

smoother. The average accuracy values increase relatively steeply for a fitness threshold between $0 \leq \Delta_{fit,max} \leq 0.03$ to reach a stable level for $\Delta_{fit,max} > 0.04$. The best average performance boost is also around 1.5 percentage points of generalization accuracy. Statistically significant changes already appear for $\Delta_{fit,max} \geq 0.028$ without any interruptions. The optimal trade-off between generalization boost and computational effort is obtained for a threshold value range of $0.03 \leq \Delta_{fit,max} \leq 0.04$. Larger values of $\Delta_{fit,max}$ do not improve the generalization accuracy anymore but increase the computational effort.

The best overall multi-pipeline classifier that is observed during all repetitions achieves a generalization accuracy of 0.9875 with $N_{Pipes} = 17$.



(a) Static selection

(b) Fitness-dependent selection

Figure 7.4: Generalization accuracy development of the multi-pipeline classifier based on the trajectory of the *ECA-full* algorithm for the *coins* dataset. The plots show the average values and the lines indicate the standard deviations. Statistically significant deviations compared to the single classification pipeline are denoted with an exclamation mark and a triangle.

7.2.5 Summary

The AROMS-Framework performs well on the *coins* dataset and significant improvements regarding the accuracy performance are achieved. The proposed standard variant *ECA-full* shows the best performance compared to all baselines and also the *Auto-WEKA* framework. It is observable that the multi-pipeline classifier extension can even further improve the generalization accuracy. The multi-configuration graph is able to produce well comprehensible visualizations. However, it has to be kept in mind that the *coins* dataset is only an artificial scenario to validate and understand the results of the framework.

7.3 Object Recognition in Steel Samples

The second classification task is denoted as *steel* dataset. It originates from the image-based cleanliness measurement system for steel samples that serves as the motivating example in the introduction of this work (see section 1.1). The process can be summarized briefly as follows. A steel sample is analyzed by slicing thin layers using a milling machine. The milled surfaces are imaged with a high-resolution scanner and objects on the surface are detected using an anomaly segmentation algorithm. The resulting objects need to be classified into three classes, namely inclusion candidates, cracks and process artifacts. Figure 1.2 shows exemplary objects. A feature-based classification approach is chosen and a set of potentially useful texture and shape descriptors is

extracted. The shape of the objects seems to be characteristic for certain classes and therefore, more shape and object contour features are used than in the *coins* dataset. The dataset contains the following feature groups (see appendix A for explanations and references of the features):

- simple shape features: area, eccentricity, perimeter, circularity, rectangularity, convexity (6 feature groups with 1 dimension each),
- function-based contour descriptors: centroid distance, centroid area, contour angle (3 feature groups with 12 dimensions each),
- statistical features of the gray value histogram: mean, standard deviation, skewness and kurtosis (4 feature groups with 1 dimension each),
- Hu moments of the gray value texture (1 feature group with 7 dimensions),
- Local Binary Patterns (LBP) gray value texture descriptors in different variants: uniform (1 feature group with 59 dimensions), rotation invariant (1 feature group with 36 dimensions) as well as uniform + rotation invariant (1 feature group with 10 dimensions) and
- low-level pixel features in form of down-scaled gray value images: 5×5 pixels (1 feature group with 25 dimensions), 10×10 pixels (1 feature group with 100 dimensions) and 20×20 pixels (1 feature group with 400 dimensions).

The complete feature set contains 20 feature groups with totally $D_{in} = 683$ feature dimensions. As the generation of ground truth annotation is time-consuming, only 748 objects with class labels are available. The class *inclusion candidate* contains 204 samples, the class *crack* contains 274 samples and the *artifact* class contains 270 samples. The dataset is randomly subdivided into 50% training dataset and 50% test dataset and this division is fixed for all experiments.

7.3.1 Central Performance Criteria

This section presents the central performance criteria of the *ECA-full* optimization algorithm for the *steel* dataset.

Accuracy Values

Table 7.6 shows the cross-validation and generalization accuracy values of the *ECA-full* algorithm compared to the two baseline methods and the *Auto-WEKA* framework. The two baseline methods are outperformed significantly with respect to both accuracy values. The baseline SVM performs poorly, as it does for the *coins* dataset. Compared to that, the baseline random forest performs better here and achieves average accuracy values that are only three

	Cross-validation accuracy	Generalization accuracy
ECA-full	**0.8188** ± 0.0132	**0.8115** ± 0.0206
Baseline SVM	0.3384 ± 0.0151 (!)	0.3663 ± 0.0 (!)
Baseline RF	0.7727 ± 0.0102 (!)	0.7856 ± 0.0157 (!)
Auto-WEKA	(not comparable)	0.8027 ± 0.0304

Table 7.6: Comparison of cross-validation and generalization accuracy results for the *steel* dataset with mean $\pm 1\sigma$. Statistically significant deviations compared to the proposed *ECA-full* algorithm are denoted with an exclamation mark.

to four percentage points lower than the results of the *ECA-full* algorithm. The *Auto-WEKA* framework performs very similar compared to the *ECA-full* algorithm and reaches an average generalization accuracy which is only around one percentage point lower.

It is a general observation that the best accuracy results of the *steel* dataset are worse than the best results of the *coins* dataset. Furthermore, the standard deviations of the accuracy values of the *ECA-full* algorithm are higher compared to the *coins* dataset, which is especially the case for the generalization accuracy. This effect can also be noticed for the results of the *Auto-WEKA* framework. These observations indicate that the real-world classification problem is harder to solve than the case study task, which is not surprising.

Best Configurations

The details of the best configurations found by the *ECA-full* algorithm for the *steel* dataset are listed in table 7.7. A similar effect compared to the *coins* dataset can be observed here: The configuration with the highest cross-validation accuracy does not necessarily generalize in an optimal way. However, the difference between the cross-validation and the generalization accuracy amounts less than two percentage points for the *steel* dataset – which is better compared to the *coins* dataset. The two best configurations shown in table 7.7 differ a lot from each other with respect to the selected features, preprocessing and feature transform algorithms. Only the classifiers are the same but with fairly different hyperparameters. This indicates that multiple algorithm combinations perform similarly well.

Optimization Runtime

The optimization runtimes for the *steel* dataset are listed in table 7.8. Compared to the *coins* dataset the average optimization runtimes of the *ECA-full* algorithm are around half an hour longer. However, this increase still seems

	Best fitness	Best generalization
Cross-validation accuracy	**0.8423**	0.8369
Generalization accuracy	0.8289	**0.8422**
Feature subset	340 features	365 features
Feature preprocessing	Standardization	Rescaling
Feature transform	NCA, D_{Trans}=25, $regularization = 1$	none
Classifier	Gaussian SVM, c_{reg}=1.97, γ_{Gauss}=1.35	Gaussian SVM, c_{reg}=5.38, γ_{Gauss}=0.09

Table 7.7: Best configurations of the *ECA-full* algorithm for the *steel* dataset with the highest fitness and generalization accuracy across the repeated experiments.

	Optimization runtime [min]	Classification time per instance [ms]
ECA-full	102.69 ± 10.73	3.60 ± 2.63
Baseline SVM	**0.05** ± 0.01 (!)	**0.79** ± 0.21 (!)
Baseline RF	0.50 ± 0.03 (!)	2.33 ± 0.97
Auto-WEKA	1,452.32 ± 33.14 (!)	(not available)

Table 7.8: Comparison of optimization runtime and classification time results for the *steel* dataset with mean $\pm 1\sigma$. Statistically significant deviations compared to the proposed *ECA-full* algorithm are denoted with an exclamation mark. Note that the classification times of the *Auto-WEKA* framework have not been logged.

to be reasonable as the *steel* dataset has a higher feature dimensionality compared to the *coins* dataset (683 vs. 642 dimensions) and the number of features has the greatest impact on the search space size (see section 5.2.2).

The baseline methods are also slightly slower compared to the *coins* dataset but, of course, magnitudes faster than the *ECA-full* optimization. The optimization runtime budget for the *Auto-WEKA* framework is 24 hours (=1,440 minutes) and the resulting runtimes are obviously significantly longer than the typical optimization runtimes of the *ECA-full* algorithm.

Classification Time

The classification times of the resulting classifiers can be found in table 7.8. The baseline SVM is the fastest classifier for this dataset. The best classification pipelines found by the *ECA-full* algorithm are the slowest but the difference to the random forest is not significant. However, an average classification time of 3.6 milliseconds per instance can still be considered as relatively fast.

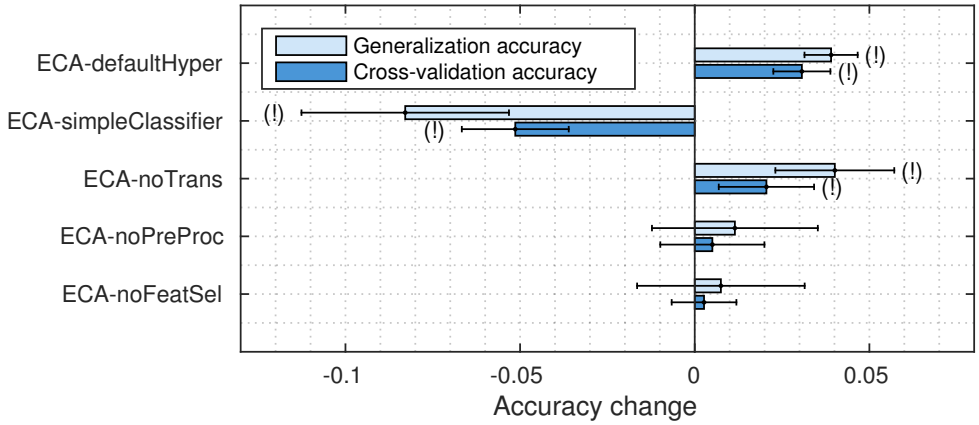


Figure 7.5: Impact of the *ECA* variants compared to the *ECA-full* algorithm on the cross-validation and generalization accuracy for the *steel* dataset. The rectangular bars show the average difference and the standard deviations are denoted through lines. Statistically significant deviations compared to the proposed *ECA-full* algorithm are denoted with an exclamation mark.

7.3.2 Impact of the Pipeline Components

This section analyzes the impact of the pipeline components with respect to the central performance criteria.

Accuracy Values

Figure 7.5 illustrates the results of the *ECA* variants regarding the cross-validation and generalization accuracy. Surprisingly, the *ECA-defaultHyper* and *ECA-noTrans* variant perform significantly better than the *ECA-full* algorithm even though certain components are restricted. Typical gains of the average generalization accuracy lie in the range of around four percentage points, which is even higher than the gain for the cross-validation accuracy values with about two to three percentage points. However, there is still a clear correlation between the cross-validation and the generalization accuracy. The best performing solution is found during one experiment repetition of the *ECA-noTrans* variant with a generalization accuracy of 0.8690.

These findings need to be investigated further because – at least in theory – the *ECA-full* algorithm could find the same configurations as the restricted variants. However, the larger search space seems to be responsible for this effect. The *ECA-full* algorithm achieves lower cross-validation accuracy values indicating that it gets stuck more likely in local optima here. It might be the case for the *steel* dataset that all feature transforms except the identity function fail and that – by chance – the standard hyperparameters of the classifiers are well performing. In this case, the restricted variants have the advantage that better solutions are found faster and thus allow a longer fine-

tuning phase. One way of coping with these obvious issues of the *ECA-full* variant is a variation of the evolutionary metaparameters. A higher number of initial individuals μ_{init} and more children λ could increase the chance that the algorithm can escape from local optima.

The impact of the classifier portfolio for the *steel* dataset is comparable to the effect for the *coins* dataset. The accuracy values significantly drop more than five percentage points when the *ECA-simpleClassifier* variant with its naïve Bayes classifier is used. The influence of the *ECA-noPreProc* and *ECA-noFeatSel* variants is less important here since the accuracy differences are marginal and not statistically relevant compared to the *ECA-full* variant.

Optimization Runtime

Figure 7.6 shows the impact of the pipeline components on the optimization runtime. Similarly to the *coins* dataset, the feature selection has a great effect on the optimization runtime, which increases more than 100 minutes on average when the *ECA-noFeatSel* variant without feature selection is used. The reason is that the optimization runtime of many algorithms depends on the feature space dimensionality and the more features are selected, the slower the algorithms become. The observation that the *ECA-noTrans* variant requires significantly less optimization runtime is also comparable to the *coins* dataset because no feature transforms with high computational complexity are considered in this *ECA* variant. The preprocessing affects the optimization runtime in a different way compared to the *coins* dataset as the *ECA-noPreProc* variant requires slightly more optimization runtime than the *ECA-full* algorithm here. A possible reason for this effect is that faster algorithms are performing better with preprocessed features for the *steel* dataset. Therefore, faster methods are evaluated more often than slower ones during the optimization process. Finally, the hyperparameter tuning and the classifier portfolio have no significant effect on the optimization runtime.

7.3.3 Impact of Selected Design Decisions of the ECA Algorithm

This section describes the impact of the improvement of the initial population, the holistic cross-validation and the early discarding system on the accuracy values (see table 7.9) and optimization runtime (see table 7.10).

Improvement of the Initial Population

The improvement of the initial population regarding the feature subset has only a small impact on the results for the *steel* dataset. Only the average generalization accuracy drops less than one percentage point when a random

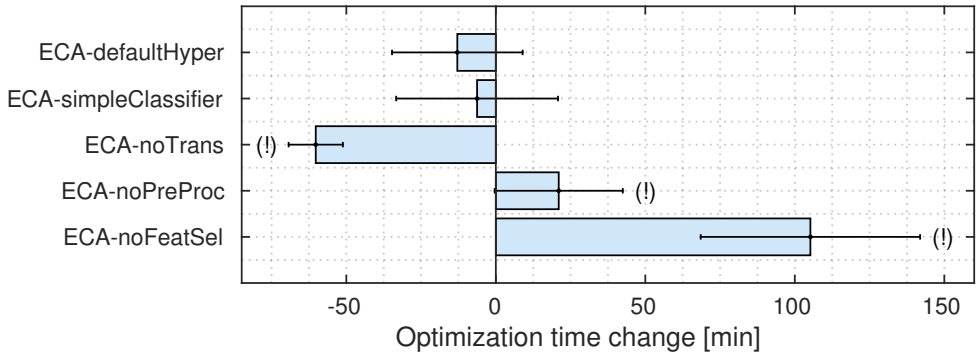


Figure 7.6: Impact of the proposed *ECA* variants compared to the *ECA-full* algorithm on the optimization runtime for the *steel* dataset. The rectangular bars show the average difference and the standard deviations are denoted through lines. Statistically significant deviations compared to the proposed *ECA-full* algorithm are denoted with an exclamation mark.

	Cross-validation accuracy	Generalization accuracy
ECA-full	0.8188 ± 0.0132	0.8115 ± 0.0206
– random initial population	0.8161 ± 0.0129	0.8053 ± 0.0153
– classifier-only cross-validation	**1.0000** ± 0.0 (!)	0.3548 ± 0.0379 (!)
– no early discarding system	0.8278 ± 0.0198	**0.8198** ± 0.0276

Table 7.9: Impact of selected design decisions of the *ECA* optimization algorithm on the cross-validation and generalization accuracy for the *steel* dataset with mean $\pm 1\sigma$. Statistically significant deviations compared to the proposed *ECA-full* algorithm are denoted with an exclamation mark.

	Optimization runtime [min]
ECA-full	**102.69** ± 10.73
– random initial population	107.31 ± 15.44
– classifier-only cross-validation	104.72 ± 8.93
– no early discarding system	388.59 ± 24.43 (!)

Table 7.10: Impact of the optimization metaparameters on the optimization runtime for the *steel* dataset with mean $\pm 1\sigma$. Statistically significant deviations compared to the proposed *ECA-full* algorithm are denoted with an exclamation mark.

initial population is used, however, the differences are not statistically relevant. The impact on the optimization runtime is not statistically noticeable as well.

The observation that additional information in form of improved feature subsets does not have any noticeable impact for this dataset can be explained as follows. Section 7.3.2 discusses that the aspect of feature selection is nearly irrelevant for the accuracy values of the *steel* dataset. This is shown by the analysis of the *ECA-noFeatSel* variant that does not perform any feature selection. Its impact on the accuracy values is not statistically relevant (see figure 7.5).

Holistic Cross-Validation

The tremendous effect of holistic cross-validation is similar to the effect for the *coins* dataset: The cross-validation accuracy of the classifier-only cross-validation is constantly at 100% for all repetitions of the experiment, while the average generalization accuracy is only around 35%. The missing generalization estimation of the feature transforms is responsible for these very poorly performing configurations when a classifier-only cross-validation is used.

Early Discarding System

The early discarding system has the same effect which is observed in the analyses of the *coins* dataset. If early discarding is not involved, the optimization process needs 6.5 hours on average – this is 3.8 times slower than with early discarding. Similar to the *coins* dataset, the accuracy results are also not affected significantly.

7.3.4 Extended Optimization Analyses

The results of the multi-configuration graphs and the multi-pipeline classifier for the *steel* dataset are presented in this section.

Multi-Configuration Graphs

Figure 7.7 shows two exemplary multi-configuration graphs for the *steel* dataset that have been obtained by two repetitions of the *ECA-full* algorithm. The graphs allow to quickly analyze that the two configuration distributions differ considerably with respect to the selected features and algorithms.

The graph in figure 7.7 (a) depicts that the combinations of the NCA and LMNN feature transforms with the Gaussian kernel SVM dominate the best solutions. The random forest also appears more often than the rest of the classifiers. Generally, the diversity of selected algorithms is relatively large for this dataset. The dark shading of a large amount of features indicates that

no clearly dominating feature subset is found by this particular optimization run.

The graph in figure 7.7 (b) shows that in this particular run combinations of the LMNN feature transform and the identity (no transform) along with the random forest classifier perform best. The diversity of algorithms is much smaller compared to the first graph. Furthermore, unlike in the first graph, the shading of the features indicates that some feature groups appear more frequently in the best solutions than others, namely the perimeter, the rectangularity, gray value features, Hu moments and the area feature. The order of the feature groups differs from the first graph as well.

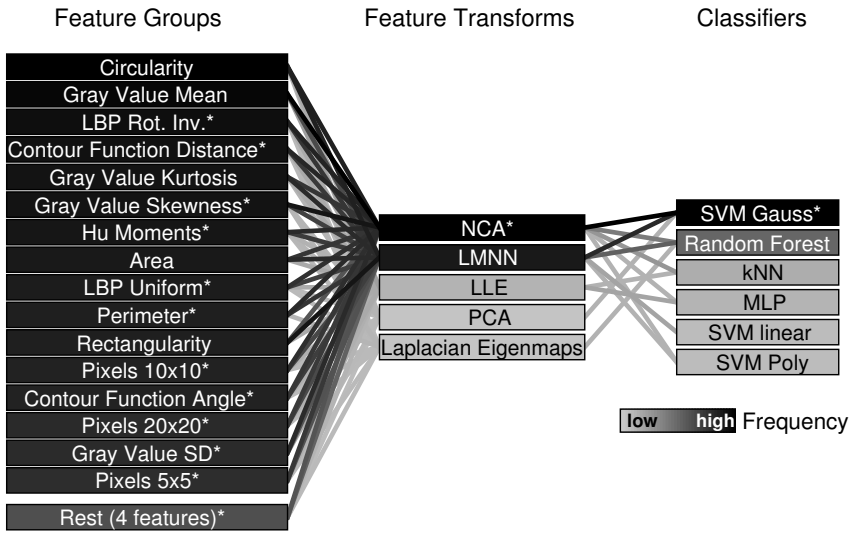
The huge differences between the two multi-configuration graphs show that not a single algorithm configuration clearly outperforms all others but that diverse algorithm configurations exist which perform almost equally well for this dataset. Therefore, it can be expected that the fitness distribution in the objective function for the *steel* dataset has more local optima and plateau-like areas with similarly high values compared to the fitness distribution of the *coins* dataset. The multi-configuration graphs for the *coins* dataset show very similar dominating feature and algorithm combinations across the repeated optimization runs. Figure 7.8 depicts how the fitness distributions for the two datasets could look like. The fitness distribution for the *coins* dataset is probably more unimodal-like, as depicted in figure 7.8 (a), while the distribution for the *steel* dataset might be similar to figure 7.8 (b).

The obviously more "difficult" fitness distribution in the objective function for the *steel* dataset may be an explanation why the *ECA-full* algorithm tends to get stuck in local optima here and is outperformed by the limited framework variants *ECA-defaultHyper* and *ECA-noTrans* (see section 7.3.2).

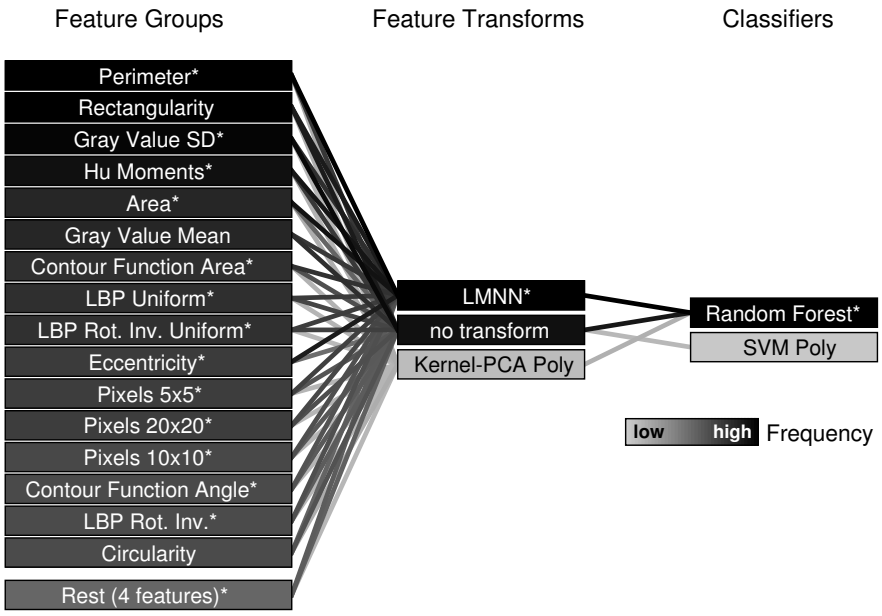
Multi-Pipeline Classifier

Figure 7.9 shows the results of the multi-pipeline classifier of the *ECA-full* algorithm for the *steel* dataset with respect to the generalization performance. The outcome of the static selection of the number of pipelines is depicted in figure 7.9 (a). The generalization accuracy increases quickly in the range of $3 \leq N_{Pipes} < 15$ pipelines, however, decreases slightly afterwards again. The accuracy boost in this "dent" in the graph is partly not statistically significant. Finally, the generalization accuracy again increases slowly for $N_{Pipes} > 30$ and the improvement becomes statistically significant once more. Compared to the performance of a single classification pipeline the maximum average improvement reaches about 2.5 percentage points. This is more than the maximum average boost for the *coins* dataset.

The results of the fitness-dependent selection of the number of pipelines is shown in figure 7.9 (b). It is noticeable that the average generalization accuracy values increase steeper and more consistently than in the static selection.

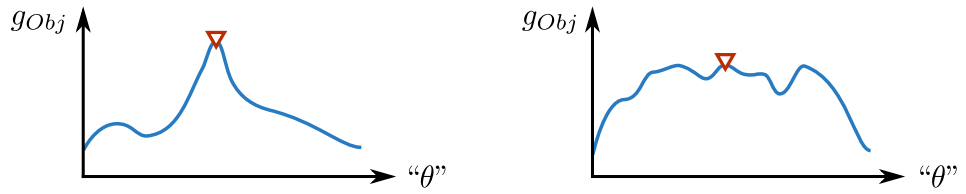


(a) Large diversity of components



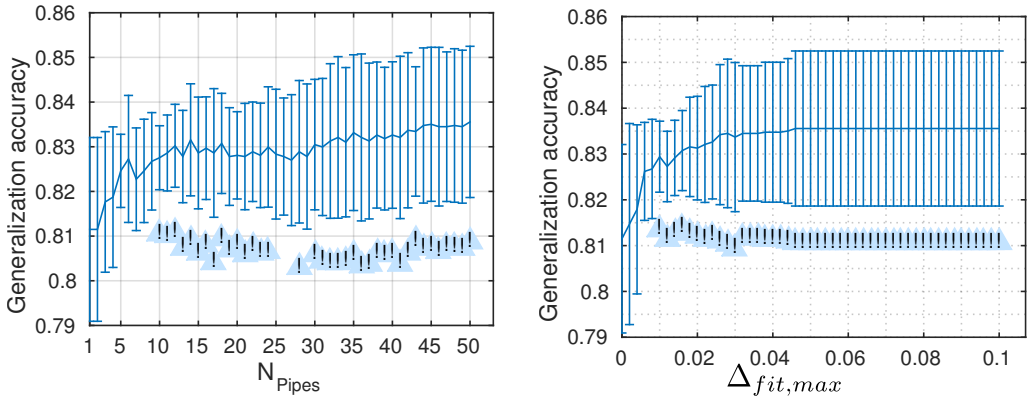
(b) Small diversity of components

Figure 7.7: Variations of the proposed multi-configuration graph of the *ECA-full* algorithm for the *steel* dataset with $N_{Configs} = 50$. The graphs are generated from two independent repetitions of the experiment.



(a) Almost unimodal objective function (b) Objective function with plateaus

Figure 7.8: Two hypothetical distributions of fitness or optimization objective functions. The global optimum is denoted with a triangle. An almost unimodal objective function (a) is expected to cause less problems in the optimization process than an objective function with many plateaus (b). Note that a realistic plot of these fitness functions is not possible as there are hundreds of variables that influence the classification pipelines.



(a) Static selection (b) Fitness-dependent selection

Figure 7.9: Generalization accuracy development of the multi-pipeline classifier based on the trajectory of the *ECA-full* algorithm for the *steel* dataset. The plots show the average values and the lines indicate the standard deviations. Statistically significant deviations compared to the single classification pipeline are denoted with an exclamation mark and a triangle.

No significant accuracy drop is observed and the changes become consistently statistically significant for all values $\Delta_{fit,max} \geq 0.01$. The maximum average generalization accuracy boost reaches around 2.5 percentage points, similarly to the static selection of the number of pipelines.

The best performing multi-pipeline classifier that is obtained during all repetitions of the experiments is found by the *ECA-noTrans* algorithm and it achieves a generalization accuracy of 0.8850 with $N_{Pipes} = 11$.

7.3.5 Summary

The AROMS-Framework performs well on the *steel* dataset compared to baseline methods and the *Auto-WEKA* framework. However, this dataset reveals interesting effects: The standard *ECA-full* variant performs well, but two other variants, one without hyperparameter tuning and one without feature transforms, perform even better regarding the accuracy results. It is suspected that a more complex fitness distribution of the objective function is responsible for more local optima. The analysis of the multi-configuration graphs with respect to the diversity of the configurations supports this hypothesis.

Finally, a pipeline configuration with a generalization accuracy value of 0.8690 is found. The multi-pipeline classifier can further improve the results and a maximum generalization accuracy of 0.8850 is obtained. These results illustrate that the chosen image-based object recognition approach for the steel measurement system is potentially problematic. Even the best performing multi-pipeline classifier shows an error rate of 11.5%, which is likely too high for a reliable and precise measurement system. The results show that it will likely be difficult to achieve a much higher accuracy value with the installed camera sensor in combination with an image-based classification approach. However, it cannot be completely excluded that the development of better features or a better preprocessing of the image data, such as noise filters, would improve the results. This aspect is also discussed in the outlook in section 8.3.3.

An alternative or additional measurement system should be considered, e.g., spectroscopy-based[6] systems, that allow a chemical analysis of the objects inside the steel. Furthermore, a better image-sensor with a higher resolution that delivers more information about the objects could be considered.

7.4 UCI Classification Tasks

This section presents the results of the AROMS-Framework on a larger amount of classification problems from different applications and research fields. The datasets are publicly available and originate from the UCI machine learning repository [Bache and Lichman, 2013]. Eleven datasets have been selected, which are listed in table 7.11. In order to save space, the datasets are referenced with their corresponding number (#) in the following tables.

The datasets have different properties with respect to the feature space dimensionality (4 – 256 dimensions), the number of training samples (150 – 1,593 samples) and the number of classes (2 – 10 classes). It is worth noting

[6]Spectroscopy systems analyze material properties in multiple wavebands of the electromagnetic spectrum (see also section 2.2.1).

#	Dataset name	Description	Dimensions	Samples	Classes
1	iris	plant classification	4	150	3
2	diabetes	medical diagnoses	8	768	2
3	breast-cancer-wisconsin	medical diagnoses	9	683	2
4	contraceptive	medication usage	9	1,473	3
5	glass	material forensics	9	214	6
6	statlogheart	medical diagnoses	13	270	2
7	australian	credit rating	14	690	2
8	vehicle	shape recognition	18	846	4
9	ionosphere	radar data analysis	34	351	2
10	sonar	material analysis	60	208	2
11	semeion-digits	text recognition	256	1,593	10

Table 7.11: Overview of the selected classification datasets from the UCI machine learning repository. Further references and information about the datasets can be obtained from [Bache and Lichman, 2013].

that the number of training samples can be considered as relatively low as datasets with millions of samples exist. However, a low number of training samples is realistic for many real-world problems.

The experiments on the UCI database are conducted in the same way as the previous datasets. They are randomly divided into 50% training and 50% test dataset while this subdivision is the same for all experiments and repetitions. In order to keep the amount of statistics at a reasonable level, only the main *ECA-full* variant of the AROMS-Framework is evaluated in detail. Note that results of the framework on these UCI datasets have also been published in [Bürger and Pauli, 2015b] and [Bürger and Pauli, 2015c].

7.4.1 Central Performance Criteria

This section discusses the central performance criteria results for the UCI datasets.

Cross-Validation Accuracy

Table 7.12 lists the cross-validation accuracy values of the baseline classifiers and the *ECA-full* algorithm on the UCI datasets. The *ECA-full* algorithm outperforms both baseline methods for all datasets with respect to this metric. Furthermore, all differences are statistically significant. However, the typical performance differences between the baseline methods and the *ECA-full* algorithm are not that large compared to the previous two datasets and usually lie in the range of one to five percentage points.

The standard deviations of the cross-validation accuracy values obtained by the *ECA-full* algorithm are remarkably low for seven of eleven datasets with values under one percent. This indicates that the *ECA* optimization

#	Baseline SVM	Baseline RF	ECA-full
1	0.9653 ± 0.0180 (!)	0.9520 ± 0.0093 (!)	**0.9893** ± 0.0056
2	0.7627 ± 0.0101 (!)	0.7445 ± 0.0078 (!)	**0.8014** ± 0.0070
3	0.9716 ± 0.0020 (!)	0.9731 ± 0.0051 (!)	**0.9834** ± 0.0020
4	0.5251 ± 0.0074 (!)	0.5275 ± 0.0078 (!)	**0.5425** ± 0.0100
5	0.6731 ± 0.0247 (!)	0.7371 ± 0.0408 (!)	**0.7995** ± 0.0258
6	0.7741 ± 0.0235 (!)	0.8156 ± 0.0158 (!)	**0.8696** ± 0.0087
7	0.6887 ± 0.0108 (!)	0.8682 ± 0.0068 (!)	**0.8758** ± 0.0076
8	0.7668 ± 0.0168 (!)	0.7448 ± 0.0098 (!)	**0.8017** ± 0.0243
9	0.9120 ± 0.0050 (!)	0.9024 ± 0.0104 (!)	**0.9512** ± 0.0081
10	0.8010 ± 0.0337 (!)	0.7895 ± 0.0325 (!)	**0.8819** ± 0.0216
11	0.9088 ± 0.0049 (!)	0.9095 ± 0.0066 (!)	**0.9297** ± 0.0044

Table 7.12: Comparison of cross-validation accuracy results for the UCI datasets with mean $\pm 1\sigma$. Statistically significant deviations compared to the proposed *ECA-full* algorithm are denoted with an exclamation mark.

#	ECA-full	#	ECA-full	#	ECA-full
1	24.09 ± 6.41	5	36.95 ± 6.06	9	28.26 ± 7.32
2	53.07 ± 12.58	6	25.82 ± 6.08	10	33.84 ± 9.08
3	53.26 ± 12.87	7	61.51 ± 12.99	11	119.71 ± 18.06
4	87.74 ± 14.93	8	54.28 ± 8.70		

Table 7.13: Optimization runtimes in minutes for the *ECA-full* algorithm on the UCI datasets with mean $\pm 1\sigma$.

algorithm is able to achieve such high fitness values consistently and that the problem of local optima seems to be less relevant for the UCI datasets.

Optimization Runtimes

The optimization runtimes of the *ECA-full* algorithm for each UCI dataset are listed in table 7.13. The average optimization runtimes lie in the range between less than 30 minutes to two hours. These optimization processes are faster than the ones for the *coins* and *steel* datasets because of the lower feature dimensionalities of the UCI datasets. A higher dimensionality is expected to increase the required optimization runtime, which the theoretical problem complexity analysis (see section 5.2.2) suggests. However, this aspect is further investigated in section 7.5.2.

The optimization runtimes of the baseline methods are not listed because all baseline methods need less than 30 seconds on average to process the datasets. The optimization runtimes of the *Auto-WEKA* framework are not listed as well because its time budget is set to 24 hours for all datasets, which is also fully used by the optimization process.

#	Baseline SVM	Baseline RF	ECA-full
1	**0.02** ± 0.01 (!)	2.27 ± 1.75	10.77 ± 14.75
2	**0.02** ± 0.00	1.32 ± 1.11 (!)	0.03 ± 0.03
3	**0.01** ± 0.00	1.33 ± 1.06	1.74 ± 3.12
4	0.06 ± 0.01	1.12 ± 0.76 (!)	**0.04** ± 0.03
5	**0.02** ± 0.00 (!)	6.61 ± 3.72 (!)	18.46 ± 11.22
6	**0.02** ± 0.00	3.96 ± 3.15 (!)	0.03 ± 0.04
7	**0.03** ± 0.01	1.83 ± 1.11	1.15 ± 2.43
8	**0.03** ± 0.01	1.88 ± 1.13	1.53 ± 3.18
9	**0.03** ± 0.01 (!)	1.31 ± 1.74	4.91 ± 6.08
10	**0.04** ± 0.01 (!)	7.46 ± 3.36	13.29 ± 10.65
11	0.69 ± 0.15 (!)	1.67 ± 0.09 (!)	**0.40** ± 0.16

Table 7.14: Classification times in milliseconds per instance for the baselines as well as the best classification pipelines found by the *ECA-full* algorithm on the UCI datasets with mean $\pm 1\sigma$. Statistically significant deviations compared to the proposed *ECA-full* algorithm are denoted with an exclamation mark.

Classification Time

Table 7.14 lists the classification times per instance of the baseline classifiers and the best classification pipelines found by the *ECA-full* algorithm. The baseline SVM leads to the fastest classifiers for nine of eleven datasets. The variation of the best classification pipelines found by the *ECA-full* algorithm regarding the classification time is relatively large. On the one hand, for the *diabetes* (#2), *contraceptive* (#4) and *statlogheart* (#6) datasets very fast average classification times under 0.05 milliseconds per instance are achieved. On the other hand, for the *glass* (#5) and *sonar* (#10) datasets the pipelines need more than 10 milliseconds on average to classify instances and are thus much slower than the baselines. The reason for this large variation of classification times and the difference compared to the baselines is certainly the incorporation of feature transforms into the classification pipelines that contribute to the overall processing speed.

Generalization Accuracy

The generalization accuracy results are listed in table 7.15. The significant performance boost of the *ECA-full* algorithm compared to the baseline algorithms that is observed for the cross-validation metric is not conveyed to the generalization performance. In fact, the *ECA-full* algorithm only performs better than the baselines for two of eleven datasets, namely for the *contraceptive* (#4) and *vehicle* (#8) dataset. The differences between the *ECA-full* algorithm and the baselines are relatively small for most datasets (one to two percentage points); except for the *sonar* (#10) dataset in which the baseline

SVM performs more than five percentage points better. However, there is no noticeably best performing baseline method, but the baseline SVM provides the lowest standard deviations.

The generalization accuracy is also compared to the *Auto-WEKA* framework. Its average generalization accuracy values are better than the *ECA-full* results for ten of eleven datasets. However, the differences are only statistically significant in five of eleven cases. When the *Auto-WEKA* framework is also compared to the baseline classifiers, it only outperforms them in five of eleven cases.

One way to explain the relatively disappointing generalization results of the *ECA-full* algorithm for the UCI datasets are possible overfitting effects. This explanation is supported by the consistently high cross-validation accuracy values during the optimization using the *ECA-full* algorithm. The adaptation of the classification pipeline is purely data-driven and does not consider regularization that could limit the complexity of the used components. These observations lead to the assumption that even holistic cross-validation is not sufficient to prevent an overadaptation to the training dataset when the huge adaptability of the proposed classification pipeline is considered. The *Auto-WEKA* framework (see section 3.2.5) comprises fewer degrees of freedom compared to the *ECA-full* algorithm and finds better generalizing solutions for the UCI datasets. However, it is also outperformed by baseline methods in six of eleven cases. Therefore, it can be stated that *Auto-WEKA* suffers from overfitting effects as well.

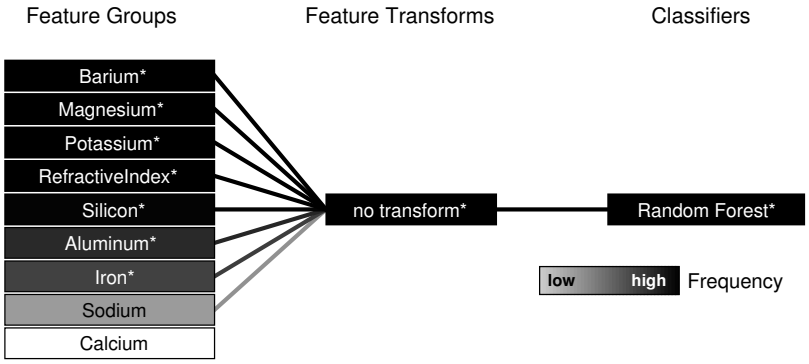
An alternative explanation is that the baseline classifiers already perform "too well" on the UCI datasets, or – in other words – that the chosen tasks of the UCI database are too simple and thus the repertoire of the machine learning solutions in the AROMS-Framework is not necessary and even counterproductive.

7.4.2 Extended Optimization Analyses

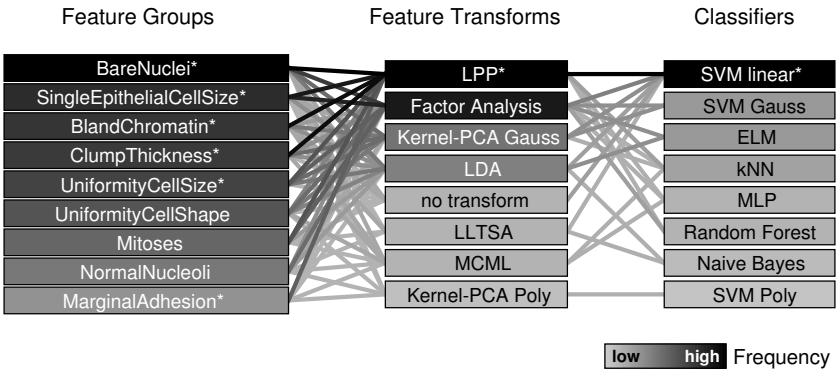
This section presents the results of the multi-configuration graph and the multi-pipeline classifier for the UCI datasets.

Multi-Configuration Graphs

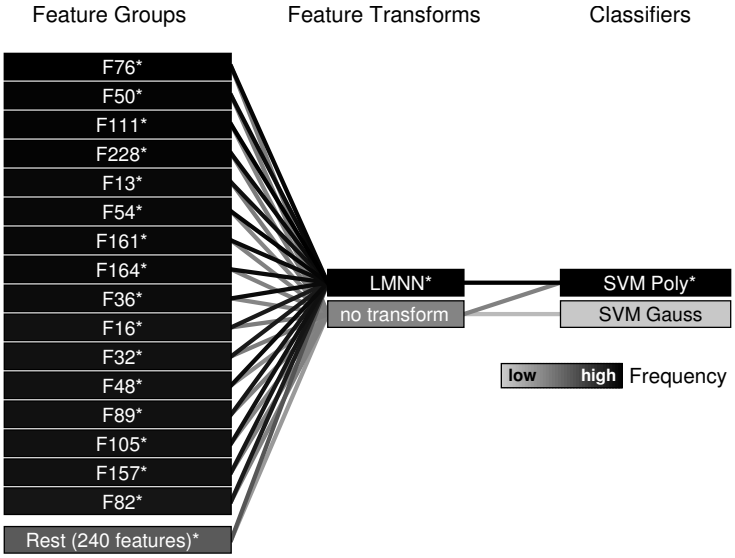
For the sake of clarity, the multi-configuration graphs are only evaluated for specific, but representative UCI datasets that have been processed with the *ECA-full* algorithm. The *glass* (#5), *breast-cancer-wisconsin* (#3) and *semeion-digits* (#11) datasets have been selected to show typical variations of the proposed graph and to explain potentials and also limitations of this visualization approach. Figure 7.10 depicts three exemplary multi-configuration graphs of the aforementioned datasets.



(a) Graph for the *glass* (#5) dataset



(b) Graph for the *breast-cancer-wisconsin* (#3) dataset



(c) Graph for the *semeion-digits* (#11) dataset

Figure 7.10: Exemplary multi-configuration graphs of three datasets from the UCI database generated by the *ECA-full* algorithm using $N_{Configs} = 50$.

#	Baseline SVM	Baseline RF	ECA-full	Auto-WEKA
1	**0.9707** ± 0.0084	0.9413 ± 0.0143	0.9547 ± 0.0228	0.9253 ± 0.0119 (!)
2	0.7396 ± 0.0 (!)	**0.7630** ± 0.0142 (!)	0.7510 ± 0.0071	0.7557 ± 0.0079
3	0.9677 ± 0.0 (!)	**0.9762** ± 0.0022 (!)	0.9639 ± 0.0031	0.9701 ± 0.0032 (!)
4	0.5570 ± 0.0075	0.5393 ± 0.0065 (!)	0.5603 ± 0.0184	**0.5641** ± 0.0127
5	0.6381 ± 0.0 (!)	0.7343 ± 0.0319 (!)	0.7000 ± 0.0354	**0.7467** ± 0.0257 (!)
6	0.7259 ± 0.0 (!)	0.8259 ± 0.0243	0.8207 ± 0.0223	**0.8370** ± 0.0 (!)
7	0.7058 ± 0.0196 (!)	**0.8741** ± 0.0061 (!)	0.8503 ± 0.0259	0.8529 ± 0.0102
8	0.7749 ± 0.0	0.7386 ± 0.0074 (!)	0.7844 ± 0.0348	**0.7981** ± 0.0176
9	**0.9657** ± 0.0 (!)	0.9474 ± 0.0114	0.9451 ± 0.0143	0.9611 ± 0.0063 (!)
10	**0.8738** ± 0.0 (!)	0.8359 ± 0.0072	0.8184 ± 0.0429	0.8447 ± 0.0228
11	0.9521 ± 0.0	0.9515 ± 0.0040	0.9506 ± 0.0057	**0.9541** ± 0.0112

Table 7.15: Comparison of the generalization accuracy results for the UCI datasets with mean $\pm 1\sigma$. Statistically significant deviations compared to the proposed *ECA-full* algorithm are denoted with an exclamation mark.

A common property of all multi-configuration graphs is that the best performing algorithm configurations can be spotted quickly due to the darker shading and the location of the items at the top. The diversity of the best configurations can be seen directly as well because it correlates with the amount of feature transforms and classifiers that appear in the graph. Furthermore, the amount of the edges between the vertices and also the complexity of the connections indicate the diversity of the configurations. The graphs in figure 7.10 (a) and (c) show a relatively small degree of diversity with a clearly best performing algorithm combination. The graph in figure 7.10 (b) displays a very large amount of diversity for feature transforms and classifiers. In this example, two feature transforms, namely LPP and the factor analysis, are almost equally often appearing in the best configurations. Two others, the kernel PCA and the LDA, also appear frequently while the other transforms in light gray may only appear rarely or even only once. The diversity of classifiers is also large, but merely the linear SVM dominates clearly.

The usefulness of the multi-configuration graph regarding the importance of the features depends on the dataset and its structure of the feature groups. The input data of the AROMS-Framework (see section 4.4) provides feature groups that aggregate features with multiple dimensions. In datasets with only relative few one-dimensional features like, e.g., the *glass* (#5) and *breast-cancer-wisconsin* (#3) dataset, each feature becomes a feature group. In this case, the corresponding multi-configuration graph (see figure 7.10 (a) and (b)) allows a direct analysis of the feature importance by analyzing the shading and the order of the feature groups.

However, datasets with many features that cannot be grouped easily complicate the feature importance analysis using the proposed multi-configuration graph. One example is the *semeion-digits* (#11) dataset which uses gray-

#	ECA-full	$MPC^{Static}, N_{Pipes} = 25$	$MPC^{Fitness}, \Delta_{fit,max} = 0.03$
1	0.9547 ± 0.0228	0.9733 ± 0.0243	**0.9787** ± 0.0210 (!)
2	0.7510 ± 0.0071	0.7542 ± 0.0077	**0.7557** ± 0.0071
3	0.9639 ± 0.0031	**0.9669** ± 0.0042	**0.9669** ± 0.0031 (!)
4	0.5603 ± 0.0184	0.5729 ± 0.0136	**0.5755** ± 0.0088 (!)
5	0.7000 ± 0.0354	**0.7190** ± 0.0169	0.7143 ± 0.0246
6	0.8207 ± 0.0223	0.8304 ± 0.0154	**0.8311** ± 0.0130
7	0.8503 ± 0.0259	**0.8547** ± 0.0083	0.8538 ± 0.0058
8	0.7844 ± 0.0348	**0.8014** ± 0.0208	0.7945 ± 0.0228
9	0.9451 ± 0.0143	0.9726 ± 0.0089 (!)	**0.9766** ± 0.0099 (!)
10	0.8184 ± 0.0429	0.8476 ± 0.0444	**0.8495** ± 0.0318
11	0.9506 ± 0.0057	0.9602 ± 0.0035 (!)	**0.9608** ± 0.0036 (!)

Table 7.16: Results of the generalization accuracy of the multi-pipeline classifier for the UCI datasets with mean $\pm 1\sigma$. Note that the *ECA-full* column contains the results of a single classification pipeline. Statistically significant deviations compared to the proposed *ECA-full* algorithm are denoted with an exclamation mark.

value images of handwritten digits that need to be classified. The size of the images is 16×16 pixels, which leads to a feature vector space with 256 dimensions. The features cannot be grouped in a more meaningful way than 256 one-dimensional feature groups. The corresponding multi-configuration graph is depicted in figure 7.10 (c) and shows 16 dark shaded feature groups with numeric feature names that represent the pixel indices and a rest vertex. The interpretation of the feature relevance is very unintuitive in this case as the two-dimensional image features are split into a one-dimensional feature vector. The overall dark shading does not help to distinguish important from unimportant features.

Multi-Pipeline Classifier

This section presents the results of the multi-pipeline classifier on the UCI datasets, which can be found in table 7.16. Only the *ECA-full* algorithm variant is considered. The static and fitness-dependent selection of the number of pipelines are evaluated with fixed metaparameters, which have lead to promising results for the previous two datasets. The static selection mode is denoted as MPC^{Static} and a fixed number of $N_{Pipes} = 25$ pipelines is chosen. The fitness-dependent selection mode is denoted as $MPC^{Fitness}$ and a fitness threshold of $\Delta_{fit,max} = 0.03$ is used.

Both variants of the multi-pipeline classifier improve the generalization accuracy compared to the single classification pipeline for all datasets. The differences between the two variants are not very large, however, the fitness-dependent selection achieves the overall best performance and wins for eight of eleven datasets. Nevertheless, only relatively few results show a statisti-

#	Best single pipeline	Best multi-pipeline	N_{Pipes}
1	0.9867	**1.0000**	12
2	0.7630	**0.7682**	27
3	0.9707	**0.9765**	5
4	0.5891	**0.5932**	29
5	0.7333	**0.7524**	4
6	0.8444	**0.8593**	8
7	0.8779	**0.8808**	3
8	0.8483	**0.8555**	37
9	0.9714	**0.9886**	37
10	0.9029	**0.9223**	3
11	0.9584	**0.9685**	15

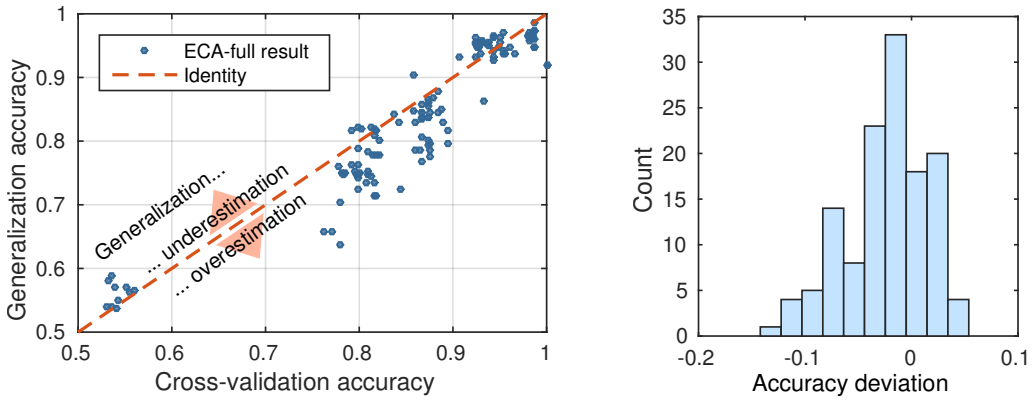
Table 7.17: Overall best results of the generalization accuracy of the multi-pipeline classifier for the UCI datasets during the repetitions of the experiments using the *ECA-full* algorithm.

cally significant improvement. The average performance improvement ranges from less than one percentage point to more than three percentage points, e.g., for the *ionosphere* (#9) and *sonar* (#10) dataset. The multi-pipeline classifier outperforms the *Auto-WEKA* framework regarding the generalization accuracy on seven of eleven datasets. However, this comparison can be considered as unfair because the solutions of *Auto-WEKA* only use single classifiers.

Table 7.17 shows the overall best performing multi-pipeline classifiers for the UCI datasets compared to the best single pipeline performance. It can be seen that there is a multi-pipeline classifier that performs better than the best single classification pipeline for all datasets. The expected performance improvement depends on the dataset and can be larger than two percentage points of accuracy. The best multi-pipeline classifier is obtained with $N_{Pipes} \approx 16.36 \pm 13.62$ pipelines on average. However, the large standard deviation indicates that the optimal number of pipelines depends on the dataset and is hard to predict.

7.4.3 Summary

The results of the AROMS-Framework on the UCI datasets are not clearly superior. The promising high cross-validation results do *not* necessarily lead to better generalization accuracy values. One explanation for the slightly disappointing results are overfitting effects. Furthermore, the UCI datasets could also be "too easy" so that standard baseline classifiers already perform optimally. However, the multi-pipeline classifier can considerably increase the generalization accuracy values for all datasets. The multi-configuration graph



(a) Cross-validation vs. generalization

(b) Deviation histogram

Figure 7.11: Comparison between the cross-validation and generalization accuracy results of the *ECA-full* algorithm for all datasets with all repetitions. The difference histogram in (b) shows the distribution of the deviations between the cross-validation and generalization accuracy.

is potentially useful for all datasets, however, an intuitive feature relevance analysis requires one-dimensional feature groups with reasonable names.

7.5 Analyses across all Datasets

This section presents comprehensive analyses and results of the *ECA-full* algorithm across all 13 datasets that are considered within this work. The central performance criteria are in focus of these analyses, namely the accuracy values, the optimization runtime and the classification time.

7.5.1 Generalization Estimation

The generalization of every pipeline configuration is estimated by the proposed holistic cross-validation method (see section 5.1.2). Figure 7.11 shows the connection between the actual generalization accuracy on the separated test dataset and the estimation of it provided by the holistic cross-validation method. All repetitions of the *ECA-full* algorithm on all datasets are considered. An ideal generalization estimation method would predict exactly the generalization accuracy based on the training dataset only. This ideal generalization estimation is denoted with the dashed identity line in figure 7.11 (a). The correlation between the estimation and the actual generaliza-

tion is clearly visible and the Pearson correlation coefficient[7] is close to one with a value of $r_{corr} = 0.9417$. However, the proposed method tends to overestimate the generalization performance as most repetitions achieve a lower generalization accuracy than predicted by the holistic cross-validation. The average difference between the cross-validation and generalization accuracy is 0.0244 ± 0.0399, which is an overestimation of more than two percentage points of accuracy on average. This can also be seen in figure 7.11 (a), in which most points lie below the identity line. Figure 7.11 (b) shows the histogram of differences between the estimation and actual generalization. The differences are fairly normally distributed, which indicates that outliers in form of very bad generalizing solutions are relatively unlikely.

7.5.2 Optimization Runtime

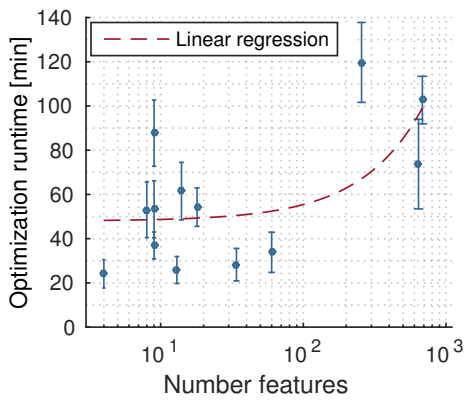
The observed optimization runtimes show a large variation that depends on the dataset. Figure 7.12 depicts plots that connect properties of all investigated datasets to the corresponding optimization runtimes of the *ECA-full* algorithm. More precisely, the impact of the input feature space dimensionality D_{in} and the number of training samples N_T are analyzed. It becomes obvious that both aspects are relevant but the optimization runtimes correlate stronger with the number of training samples ($r_{corr} = 0.8135$) than with the feature space dimensionality ($r_{corr} = 0.5989$). A precise prediction function of the optimization runtime for a given dataset cannot be made as too many non-deterministic and implementation-specific effects would need to be considered. However, the optimization runtime is roughly correlated to $\mathcal{O}(N_T + D_{in})$.

7.5.3 Classification Time

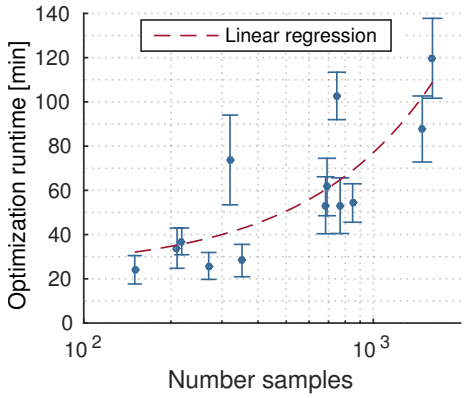
The classification times of the resulting best classification pipelines found by the *ECA-full* algorithm are heavily depending on the dataset. The distribution of the classification times is depicted in figure 7.13 while all repetitions of the *ECA-full* algorithm on all datasets are considered. The average classification time per instance is 4.32 ± 8.40 milliseconds. However, the values are obviously not normally distributed as no single peak can be spotted in the histogram.

Figure 7.14 depicts the connection between dataset properties and the classification time. Figure 7.14 (a) shows that the classification time is slightly negatively correlated ($r_{corr} = -0.2089$) with the number of features D_{in}. This

[7]The Pearson correlation coefficient measures the linear correlation between two variables and lies in the range of $r_{corr} \in [-1, +1] \subset \mathbb{R}$ [Howell, 2006]. Values of r_{corr} close to -1 or 1 denote a strong linear correlation while values closer to zero indicate a weaker correlation.



(a) Feature space dimensionality



(b) Number of training samples

Figure 7.12: Influence of dataset properties on the optimization runtimes of the *ECA-full* algorithm for all datasets. The dots denote the average optimization runtimes and the vertical lines indicate the corresponding standard deviations. Note that the linear regression appears as a non-linear curve due to the logarithmic scale of the abscissa.

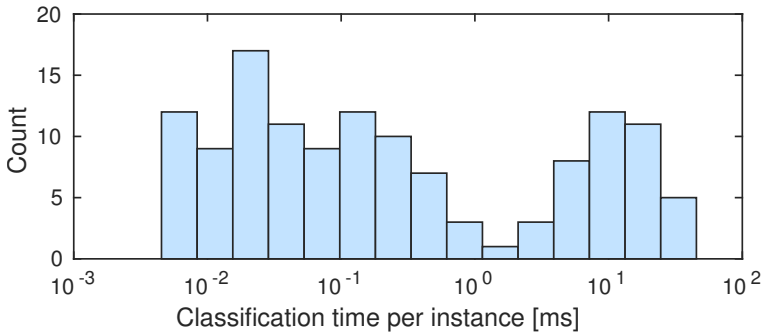
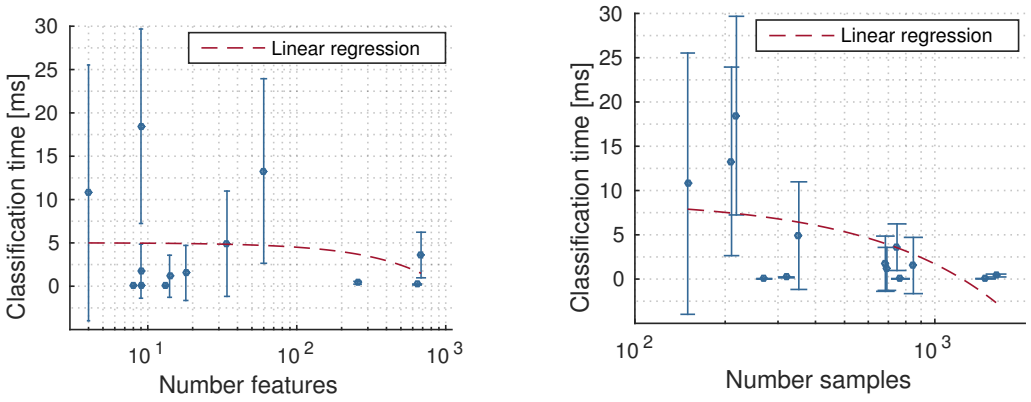


Figure 7.13: Histogram of observed classification times in milliseconds per instance of classification pipelines found by the *ECA-full* algorithm for all datasets and all experiment repetitions.

is surprising because it could be expected that a higher feature dimensionality would lead to classifiers that require more time for the classification of instances. However, the observed weak negative correlation is most likely a random effect as there is no theoretical reason for the general trend that a higher feature dimensionality leads to faster classifiers.

Figure 7.14 (b) shows that the classification time is also slightly negatively correlated ($r_{corr} = -0.5648$) with the number of training samples N_T. It could have been expected that the classifiers tend to learn a more complex model when more training instances are available and thus would need more classification time – which cannot be observed. However, the negative



(a) Feature space dimensionality (b) Number of training samples

Figure 7.14: Influence of dataset properties on the classification time in milliseconds per instance of classification pipelines found by the *ECA-full* algorithm for all datasets. The dots denote the average classification times and the vertical lines indicate the corresponding standard deviations. Note that the linear regression appears as a non-linear curve due to the logarithmic scale of the abscissa.

correlation needs to be explained with random effects as well as it lacks a theoretical background.

The analyses reveal that the classification time is *not* correlated with these two central dataset properties in a meaningful way – unlike the optimization runtime. The actual processing and classification is much faster than the training phase of the algorithms and less dependent on the training dataset properties. The actual speed of any algorithm is also heavily dependent on its implementation and many algorithms provided by Matlab are not optimized for speed. Furthermore, the *ECA-full* algorithm selects these algorithms solely depending on their classification performance and *not* based on the classification speed[8], which explains the observed random negative correlations.

7.6 Discussion

This chapter presents the evaluation of the proposed AROMS-Framework on multiple datasets with respect to multiple criteria. The findings across the different scenarios and datasets are discussed in this section.

[8]This aspect is discussed in the outlook, which can be found in section 8.3.2.

7.6.1 Performance of the ECA-full Algorithm

The evaluation is focused on the *ECA-full* optimization algorithm, which is the most "interesting" variant of the AROMS-Framework. Compared to the baseline classifiers, it achieves the highest average cross-validation accuracy values for almost all datasets. Also the corresponding standard deviation values are at a relatively low level. Thus, the proposed optimization algorithm can relatively consistently achieve proper solutions – at least with respect to the cross-validation accuracy.

However, classification pipelines with the "best" configurations found by the *ECA-full* algorithm do not necessarily lead to the best generalization accuracy values on the separated test dataset. This is especially the case for the UCI datasets in which the baseline classifiers partially show a better generalization performance, even though their cross-validation accuracy values are lower. It can be expected that overfitting effects, which occur due to the large degree of adaptability of the classification pipeline, are responsible for these results. The proposed holistic cross-validation method tends to overestimate the generalization accuracy by around two percentage points but still shows a strong correlation between the estimated and the actual generalization accuracy.

The typical optimization runtimes of the *ECA-full* algorithm for the investigated datasets range between less than thirty minutes up to two hours, which can be considered as surprisingly fast. A naïve grid search approach is expected to require billions of years for many of the datasets, mainly due to the exponential complexity of the feature selection problem (see section 5.2.2). It turns out that the proposed extended Evolution Strategies are especially suitable to handle the feature selection efficiently in order to find good feature subsets. However, the experiments show that the number of samples of the training dataset and also the feature space dimensionality do have a moderate impact on the optimization runtime – but there is certainly no exponential increase of computation time. Finally, it can be stated that the *ECA-full* algorithm is 12 to 48 times faster than the *Auto-WEKA* framework, while a comparable classification performance is achieved.

The average classification times of the resulting pipelines found by the *ECA-full* algorithm tend to be greater compared to baseline classifiers but instances are usually classified in less than 0.1 – 20 milliseconds. The optimal classification pipelines may also be as fast as or even faster than the baseline methods. However, the classification speed depends on the algorithm selection in the pipeline that itself depends on the actual learning task. It can be expected that the classification time is nearly impossible to predict before the actual optimization process. The *ECA* algorithm only considers the cross-validation accuracy values in the optimization process. This means that if

slower methods perform better, the classification speed will automatically be lower as well.

7.6.2 Impact of the Pipeline Components

The analysis of the impact of the classification pipeline components reveals that all investigated components, namely feature selection, feature preprocessing, feature transform, classifier portfolio and hyperparameter tuning are *potentially* useful to improve the accuracy values. However, the *steel* dataset shows that the full amount of adaptability of the *ECA-full* algorithm can actually lead to worse accuracy values. This can be explained due to a larger amount of local optima in the search space that disturb the optimization process.

The analysis of the optimization runtimes reveals that basically two pipeline components have a significant impact. The optimization process becomes faster when the feature selection is used because the data dimensionality is decreased. In contrast to that, the optimization runtimes increase when feature transforms are involved. Many feature transforms have complex computational models which need a lot of time to be trained.

7.6.3 Impact of Selected Design Decisions of the ECA Algorithm

It is shown that the holistic cross-validation method is absolutely necessary to estimate the generalization of the classification pipeline. The classifier-only variant of cross-validation produces configurations with "perfect" cross-validation accuracy values but actually a poor generalization performance. This illustrates that the generalization of the feature transform *must* be estimated as well as the generalization of the classifier. The impact of the holistic cross-validation on the optimization runtime is marginal, even though the computational complexity is higher.

The early discarding system turns out to be very useful as well. The required optimization runtimes decrease tremendously by a factor of almost four, while the accuracy values are not affected significantly. It can be expected that this speedup factor is closely linked to the choice of the number of cross-validation rounds N_{CV}: The aggressive discarding rules will stop the cross-validation process after one round for the vast majority of configurations. So the amount of saved cross-validation rounds will be even greater for $N_{CV} > 5$ as a greater percentage of rounds will be skipped. However, a larger number of cross-validation rounds will increase the runtimes regardless if the early discarding system is used or not: At least a fraction of the configurations requires the evaluation of all cross-validation rounds.

The improvement of the initial population regarding the feature subset turns out to be useful as well, but its impact is relatively small. The accuracy values increase slightly while the optimization runtime is not affected in a significant way. However, other improvements of the initial population regarding, e.g., the selection of algorithms or hyperparameter values, could potentially be beneficial as well.

7.6.4 Multi-Configuration Graph

The evaluation of the multi-configuration graph demonstrates that this visualization technique is suitable to support users of the AROMS-Framework to understand the best performing algorithm combinations consisting of feature transforms and classifiers very quickly. The graph can also be used to directly analyze the relevance of the features when the feature groups of the dataset have a meaningful structure. An advise to obtain such a reasonable structure is the use of multiple feature groups with reasonable names. A single, high-dimensional feature without subdivision into groups should be prevented.

Additionally, the proposed multi-configuration graph is a useful tool to estimate the diversity of well performing algorithm combinations for a given dataset. A quick view reveals the amount of vertices and edges for the feature transforms and classifiers. The more components there are, the higher the diversity is. This information allows to draw conclusions about the optimization trajectory as well as the "fitness landscape" of the objective function (see figure 7.8). Evidence is found that a high configuration diversity leads to a fitness distribution with many plateau areas and more local optima. Such difficult fitness distributions may be the reason why the proposed *ECA-full* algorithm – and potentially other optimization algorithms as well – produce suboptimal results.

7.6.5 Multi-Pipeline Classifier

Compared to a single classification pipeline the multi-pipeline classifier is able to significantly improve the generalization performance. The average generalization accuracy boost can be larger than three percentage points so that all baselines and also the *Auto-WEKA* framework are outperformed regarding the generalization. However, it has to be kept in mind that the multi-pipeline classifier comes with the disadvantage of a higher computational cost for classifying instances compared to the single classification pipeline. But the actual classification time of a multi-pipeline classifier can be reduced by a parallel processing of the pipelines as they are independent from each other.

The results proof that the fitness-dependent selection of the number of pipelines shows the best results as the optimal metaparameter value of the

fitness threshold is less dependent on the dataset than the static selection of the number of pipelines. Reasonable standard values, namely $N_{Pipes} = 25$ for the static selection and $\Delta_{fit,max} = 0.03$ for the fitness-dependent selection of the number of pipelines, are found that perform well for all investigated learning tasks.

Chapter 8

Conclusions

This final chapter concludes the whole work and proposes future improvements. It is organized as follows. Section 8.1 summarizes the work including the investigated problems, the selected approaches and the conducted experiments. Section 8.2 discusses the results with respect to the goals that have been determined in the introduction chapter. Finally, in section 8.3 an outlook for improvements of the AROMS-Framework and future work in this research field are provided.

8.1 Summary

The general topic of this work was the automatic improvement of machine learning systems for classification tasks. At first, the typical challenges that occur in machine learning have been analyzed and discussed. The most common challenges can be summarized as follows. Many problems that arise from the feature data are related to the curse of dimensionality. The selection of suitable machine learning algorithms for a given task is another great challenge that is known as the no-free-lunch theorem. Furthermore, the hyperparameters of the involved algorithms need to be tuned for each task. Additionally, the generalization of a classifier has to be estimated adequately to select a well performing one that does not suffer from overfitting. Ultimately, all these challenges occur in an arbitrary combination for most real-world classification problems.

Based on the analysis of challenges a set of suitable solutions was discussed. These solutions can be subdivided into established remedies such as feature and model selection or hyperparameter tuning. Another recently evolving solution is representation learning, which is one field of deep learning. Representation learning can be generalized with feature transforms that are learned with data. The goal of such feature transforms is that the resulting features should be better suitable for the machine learning algorithm than the original ones.

The great number of solutions leads to the consequential problem to automatically find the best suitable combination of approaches for a given learning task because a manual selection is clearly infeasible. The central contribution of this work was the AROMS-Framework that should solve this problem in a holistic way. A classification pipeline was proposed that consists of four pipeline elements with different processing tasks: the first one is responsible for feature selection, the second one for feature preprocessing, the third one for feature transforms and the last one for the actual classification. Each pipeline element has different degrees of adaptability, such as the selection of a feature subset, the selection of algorithms and hyperparameters. All hyperparameters of the pipeline are collected in the pipeline configuration which has to be adapted for each learning task.

In order to archive a fully automated configuration adaptation for each learning task, an optimization algorithm with a suitable objective function had to be developed. The objective function needs to estimate the generalization performance of the whole classification pipeline. Therefore, the holistic cross-validation method was developed. An early discarding system was introduced to increase its computational efficiency. A further analysis of the value distribution of the objective function revealed a complex interplay between the selected algorithms and hyperparameters that leads to a large amount of local optima. Furthermore, the complexity analysis of the configuration adaptation problem showed that any naïve approach like grid search is clearly infeasible.

An Evolutionary Algorithm has been selected to solve the configuration adaptation problem because these kinds of algorithms can cope with complex objective functions and can easily take advantage of parallel processing. The Evolution Strategies approach has been extended to handle the heterogeneous hyperparameter types that occur in the pipeline configuration, such as real and integer numbers, Booleans and categorical variables. The proposed *ECA* optimization algorithm uses these extended Evolution Strategies to adapt the pipeline configuration. However, the extended Evolution Strategies have been introduced in a generalized way so that this approach can be applied to other optimization problems as well.

A challenge that arises in the configuration adaptation problem is that the set of active hyperparameters depends on the selection of algorithms. A solution for this challenge was motivated with the genotype-phenotype distinction in genetics: The *ECA* algorithm appends all hyperparameters of all algorithms to the genotype. However, only those hyperparameters that belong to the selected algorithms are actually influencing the resulting classification pipeline, which is the phenotype in this case. However, the full genotype, including the information of active and inactive hyperparameter values, is considered in the evolutionary operators and is recombined, mutated and transferred to the next generation of individuals.

Additionally, the information of the optimization trajectory of the *ECA* algorithm was exploited in two ways. First, the multi-configuration graph visualizes the set of the best configurations to allow a fast understanding of the best features and algorithm combinations for a specific dataset. Secondly, the best configurations are used to set up a multi-pipeline classifier with a better generalization performance.

The evaluation of the AROMS-Framework covered multiple aspects: At first, multiple classification tasks have been selected from different application fields, such as image-based object recognition, medical diagnostics and forensics. Secondly, multiple metrics such as cross-validation and generalization accuracy as well as optimization runtime and classification time were compared. Thirdly, several variants of the *ECA* optimization algorithm and design decisions of it were evaluated to understand the internal framework functionality and to quantify effects of overfitting and local optima.

The results of the extended optimization analyses, namely the multi-configuration graph and the multi-pipeline classifier, were discussed in detail as well. The influence of randomness of the chosen optimization algorithm was considered with repeated experiments and suitable statistical tests to detect significant differences between the algorithm variants.

Finally, the source code of the Matlab implementation of the AROMS-Framework was fully published on *GitHub* [Bürger, 2016] so that other researchers can benefit from the system and conduct their own experiments.

8.2 Discussion of the Results

This section discusses the results of this work with respect to the goals that have been defined in the introduction in section 1.2.

8.2.1 Development of a Holistic Framework

An important aim was the development of a holistic framework that contains a large set of approaches to typical machine learning challenges including the field of representation learning. This goal was successfully achieved with the AROMS-Framework and its central classification pipeline. The proposed pipeline consists of four pipeline elements that comprise potentially useful processing steps for machine learning. The field of representation learning plays a central role in form of feature transforms that should improve the feature representation. The adaptation of the pipeline configuration to a given dataset is done automatically by the *ECA* optimization algorithm.

8.2.2 Improvement of the Classification Performance

Of course, the whole effort of optimizing a complex processing pipeline with numerous algorithms is only of value, if the classification performance is increasing noticeably. This goal was partly achieved as the AROMS-Framework can potentially generate significantly better generalizing classifiers. The benefit was larger on more difficult and high-dimensional datasets.

However, the AROMS-Framework did not always find solutions that generalize better than other, less time-consuming methods. Two effects were observed that are responsible for these issues: First, the proposed framework partly suffers from overfitting effects, which arise from the purely data-driven approach to optimize a highly adaptable classification pipeline to a given training dataset. Even the holistic cross-validation method could not prevent overfitting completely. Secondly, some results of the experiments showed that the *ECA-full* optimization variant, which comprises the full adaptability of the classification pipeline, can terminate in local optima and thus the found configurations achieve an inferior performance. This problem arises from certain dataset characteristics in combination with eventually suboptimal metaparameter values of the AROMS-Framework. It can be expected that especially the values of the number of initial individuals (μ_{init}) and the number of children in each generation (λ) should be increased for these difficult datasets to minimize the risk to get stuck in local optima.

The proposed multi-pipeline classifier kept its promise to increase the generalization accuracy. The evaluation showed that this was the case for all datasets that have been considered. However, its disadvantage is a higher computational effort during the classification phase. The time constraints are different for each application, e.g., a real-time object detection system in an autonomous vehicle may require a decision within a few microseconds, while a medical diagnose system might be fast enough with one decision per second. However, due to the independence of the classification pipelines, the proposed multi-pipeline classifier can benefit from parallel processing.

It was also shown that unavoidable, non-deterministic effects of the *ECA* optimization algorithm and some computational models, e.g., random forests, lead to fluctuations of the accuracy values. When the classification task at hand requires extensive tuning and a tenth of a percentage point of accuracy is important, it can be useful to repeat an optimization process with the AROMS-Framework and pick the best solution. The AROMS-Framework implementation directly supports this as the number of repetitions can be determined and the results are automatically analyzed afterwards.

8.2.3 Optimization and Classification Efficiency

The goal of a reasonably fast optimization process was achieved. Depending on the dataset the typical optimization runtime of the *ECA* algorithm – running on a relatively normal workstation computer – is within a range of minutes up to a few hours. This amount of time is tolerable as the optimization process works completely unsupervised and can run, e.g., during the night. The proposed extended Evolution Strategies were suitable to solve the highly combinatorial configuration adaptation problem. The feature selection problem, which contributes the most to the immense size of the search space, was efficiently tackled using a bitstring representation. The evaluations showed that the number of features only has a little impact on the optimization runtime. Furthermore, the proposed early discarding system has to be mentioned which was a significant contribution to the computational efficiency as well.

It was shown that the optimization runtime is correlated to the number of training samples and also the dimensionality of the training dataset. This has to be considered when datasets with more than around thousand samples should be analyzed. It can be expected that a time budget of 24 hours of optimization runtime will be sufficient for datasets with approximately 20,000 training samples. There are two ways to deal with even larger datasets: At first, it is possible to select a random subset of training samples so that the total number of training samples comes to a reasonable level. Secondly, the algorithm portfolios can be restricted to the most efficient methods only: A tremendous processing speedup can be expected when the set of feature transforms only contains fast methods, e.g., PCA, or the set of classifiers only considers classifiers that work well with large training datasets, e.g., good implementations of the SVM or random forests.

Even though the classification pipeline may use complex feature transforms and classifiers, the average classification time stayed reasonable within a range of less than 0.1 up to 20 milliseconds per instance. However, the tolerable classification time depends on the actual application; real-time systems or applications with excessive amounts of instances that need to be classified might require faster classifiers. It can be expected that the classification time will be improved when the algorithm portfolios only contain fast methods, such as linear classifiers or feature transforms.

8.2.4 Useful Statistics

A secondary goal was the generation of helpful statistics to understand the complex results of the AROMS-Framework. This aim was achieved with the multi-configuration graph that uses by-products of the optimization process. This visualization technique turned out to be an intuitive way to analyze the interplay of the most relevant features and algorithms. On the one hand,

this graph offers developers of machine learning systems the chance to quickly identify and improve the most promising approaches to their specific problem. On the other hand, this graph might also be helpful to improve the AROMS-Framework and its optimization algorithms itself because, e.g., trajectories of different algorithm variants can be compared quickly.

8.2.5 Development Effort and Required Expertise

The goal of reducing the development effort for classification systems was achieved in the sense that the AROMS-Framework provides a high degree of automation. The *ECA* optimization algorithm does not require manual intervention and the extended optimization analyses and statistics are performed and generated automatically.

The framework has numerous metaparameters, e.g., the evolutionary metaparameters, but all of them have justified standard values which work reasonably well for a wide range of tasks. The standard variant of the optimization algorithm (*ECA-full*) adapts the full set of components and it was shown that it works fairly well for most tasks. Consequently, the chances are high that even inexperienced users obtain a well working classifier using the AROMS-Framework without any manual metaparameter adaptation.

However, the no-free-lunch theorem also holds for the AROMS-Framework. There is no guarantee that the standard variant and metaparameters will deliver the best possible results. It was shown that for some datasets the standard *ECA-full* variant did not lead to the best overall performance, but, e.g., a restriction of the feature transforms actually improved the results even further. So it can be worth trying some different metaparameters and variants as well as baseline classifiers. The implementation of the AROMS-Framework directly supports the systematic analysis of metaparameters, variants and baseline methods. To give an example, most statistics and comparisons in the evaluation chapter 7 have been generated automatically.

8.2.6 Understanding of the Framework Functionality

One of the last goals was dedicated to contribute to the understanding of the internal functionality of the framework, which is obviously complex. A central task within this context was the quantification of the impact of the different pipeline components, namely feature selection, feature preprocessing, feature transforms, classifiers and hyperparameter tuning. Experiments revealed that all components of the proposed classification pipeline can potentially contribute to an improvement of the classification performance. However, the specific success of each component depends on the dataset. This result was partially expected because it is obvious that not all learning tasks necessarily suffer from, e.g., the curse of dimensionality.

The Role of Representation Learning

The experiments have shown that the incorporation of representation learning in form of feature transforms did not guarantee a generalization accuracy boost and thus did not perform "miracles". The generalization performance was even degraded for some datasets while the optimization runtime was rising significantly due to the more complex computational models behind the feature transforms. Possible explanations for the quite disappointing role of representation learning in the AROMS-Framework are the following.

First, the used datasets might contain too few training samples and too much noise. Many manifold learning techniques work well on artificial data, but they are very sensitive to noise and fail to build reasonable models. The main reasons are numerical instabilities and unsuitable assumptions of the data distribution. These issues are also reported in the results of [Van der Maaten et al., 2009]:

> "Taken together, the results of our experiments indicate that, to date, nonlinear dimensionality reduction techniques perform strongly on selected datasets that typically contain well-sampled smooth manifolds, but that this strong performance does not necessarily extend to real-world data. [...] From the results obtained, we may conclude that nonlinear techniques for dimensionality reduction are, despite their large variance, often not capable of outperforming traditional linear techniques such as PCA."

Secondly, the feature transform methods used in this work only consider one-dimensional representations of the data. However, in case of image-based object recognition, the data is inherently two-dimensional and this additional information is certainly important for machine learning. The field of deep learning also comprises artificial neural networks with two-dimensional feature data. The so-called *convolutional neural networks* (CNN) [LeCun and Bengio, 1995] use multiple layers of image data filters and operations that are inspired by receptive fields[1] in the visual system of humans and mammals. Deep CNN perform very well in image-based applications of machine learning [Krizhevsky et al., 2012]. Furthermore, CNN show a superior performance in other fields such as speech recognition [Bengio et al., 2013]. This success certainly motivates further research into the direction of deep learning. Deep neural networks, especially CNN, are also controlled by numerous hyperparameters such as the number and type of the layers as well as image filter parameters. Therefore, it can be expected that the challenge of automatic hyperparameter optimization of deep learning algorithms will become more

[1] Receptive fields are responsible to process small but overlapping areas of the perceived two-dimensional information in the visual system.

emerging in the future. The recent work of [Hutter et al., 2015] points into this direction with considering model-based optimization for deep learning.

8.2.7 Final Summary

The AROMS-Framework was the result of a holistic approach to many common challenges in machine learning. It should be seen as a useful tool to support and speedup the development process of machine learning systems. The framework supports fast feasibility studies to verify if a machine learning approach is a good option for a specific application.

However, it is certainly not the case that machine learning experts are no longer needed to develop classification systems. Due to the no-free-lunch theorem it is not guaranteed that even the most sophisticated optimization methods in combination with the best machine learning algorithms will achieve the desired performance. The design of task-specific feature descriptors will require the expertise and creativity of developers as well.

Furthermore, the success of any machine learning system will always depend on the quality and amount of the training data. The input data must contain enough information to allow a proper class distinction and should represent a great spectrum of possible variations that appear in the planned application. Otherwise overfitting effects or the representational bias will almost surely degrade the generalization performance.

8.3 Outlook and Future Work

It can be stated that most of the planned goals have been achieved by the AROMS-Framework in a satisfactory way. However, the proposed framework can still be extended and improved. Some of the most promising ideas for future work are discussed in the following.

8.3.1 Generalization Improvements

A further improvement of the generalization performance is certainly one of the most desirable aims for any machine learning optimization framework.

The *generalization estimation* is one of the most promising aspects that can be improved. This work introduced the holistic cross-validation method that already considers the generalization of the whole classification pipeline. Even though the experiments illustrated that this method is absolutely necessary to obtain good configurations, it was also shown that the method still tends to overestimate the generalization performance.

It is known that the N_{CV}-fold cross-validation method has limitations when the sample size is small. The subdivisions into disjoint training and

validation datasets reduce the actual size of the training dataset for the classifier – by a factor of $1/N_{CV}$, to be precise. In order to overcome this effect, the *bootstrap method* [Efron, 1982] could be used that resamples the training datasets and generates new "pseudo" training elements.

However, the computation time of the generalization estimation should remain reasonable as it can be expected that more sophisticated methods will be more complex. It would make sense to perform any advanced estimation method only on the most promising solutions during the optimization to save computation time.

Another effect of the AROMS-Framework was noticed with respect to non-deterministic algorithms, e.g., random forests. The current implementation only stores the configuration, but not the actual trained models and model parameters. To set up a classification pipeline for the classification mode, all algorithms and the corresponding models need to be trained again using the training data. Even though the same configuration and hyperparameters are used, it is likely the case that – at least slightly – different models are learned due to random effects of non-deterministic algorithms. These differences could generate classifier models that perform worse in the classification phase than in the optimization phase. A solution for this problem would be to *store the actual models* with all their learned internal model parameters. However, an "intelligent" model storage system will be necessary to keep a reasonable additional memory consumption.

8.3.2 Improvement of the Optimization Algorithm

The configuration adaptation problem requires a sophisticated optimization algorithm, which surely has the potential to be improved.

A central limitation of the proposed *ECA* optimization algorithm is its consideration of only *one* objective metric, namely the average cross-validation accuracy. However, many real-world applications often require that several criteria are considered. These criteria can be related to machine learning quality metrics such as the precision and accuracy values of a specific class. Other criteria, such as the classification time or the memory consumption, may also be important for real-time applications on embedded systems.

A *multi-objective optimization* or *Pareto optimization* is the right approach to simultaneously optimize several metrics. The result of such an optimization process is a set of the best solutions – called Pareto front – because usually not all criteria can be maximized simultaneously. One way to select the best solution is to simply let the user decide which solution should be picked. An example decision could be between a classifier with a cross-validation accuracy of 0.95, which needs 0.1 seconds to classify an instance, or one with an accuracy of 0.93, which only needs 0.001 seconds per instance. An

alternative to a manual selection is the use of aggregation functions that fuse multiple metrics to a single one.

Once a multi-objective optimization is realized, it would be tempting to optimize multiple criteria to improve the generalization performance of the pipeline. One additional criterion could be the consideration of the *standard deviation of the accuracy values* during the cross-validation process rather than only using the average value. The usage of this standard deviation value – which is readily available – would introduce an estimation of the classifier variance in the sense of the bias-variance dilemma. If two configurations with a similar average cross-validation performance are found, the one with a lower variance should be preferred.

Another promising criterion to be used in multi-objective optimization could be a *simplicity estimation* of configurations that acts as a kind of complexity regularization inspired by the principle of Ockham's razor[2]. A "simple configuration", e.g., a linear classifier without a feature transform, is less prone to overfitting and should be preferred over highly non-linear methods if the accuracy levels are similar.

The optimization algorithm itself could also be made more "intelligent". One way would be to automatically adapt the metaparameters of the *ECA* algorithm (see appendix E), e.g., the number μ of surviving individuals in the population, to each task or even online during the optimization process. The work of [Kramer, 2008] and [Hamadi et al., 2011] discusses promising approaches towards *intelligent and self-adapting optimization algorithms*.

Furthermore, alternative optimization algorithms could be applied to the configuration adaptation problem. One idea is a hybrid optimization algorithm which combines an Evolutionary Algorithm with *tabu search*. The information about poorly performing algorithms or hyperparameters during the optimization process could be used to label parts of the search space as unsuitable. The hope would be to save computation time on less promising solutions by not "visiting" and evaluating the labeled areas of the search space again. However, there is also the risk of excluding good algorithms prematurely, e.g., because of suboptimal hyperparameters or an unsuitable feature subset.

Ultimately, a *Bayesian or model-based optimization approach*, e.g., Sequential Model-Based Optimization (SMBO), is also promising for the configuration adaptation problem. The results of the *Auto-WEKA* framework – which have been compared to the results of the AROMS-Framework – indicate the suitability of model-based optimization.

[2]Ockham's razor is a philosophical paradigm that simple solutions should be preferred when they perform equally well than more complex ones. It is attributed to William of Ockham (ca. 1287 – 1347) who made use of this principle in his work.

8.3.3 Application Field Extension

The AROMS-Framework was designed for the application of supervised classification tasks for numeric feature vectors. The training phase works offline, which means that only a single training dataset is considered before the optimization process and every change in the training data requires a completely new optimization process. The following subsections discuss potential extensions of the application field of the AROMS-Framework.

Simultaneous Image Processing Optimization

The motivation of this work was an image-based measurement system for the cleanliness of steel, which was introduced in section 1.1. The results of the AROMS-Framework on a dataset that originates from this system were not fully satisfactory. One reason for this is certainly that the image processing steps before the actual classification of the objects should be optimized as well. The field of image processing (see, e.g., [Gonzalez and Woods, 2002]) offers almost limitless options and alternatives to perform

1. *image preprocessing* such as image smoothing filters to remove noise,
2. *object segmentation* to localize objects and extract their shape and
3. *object feature extraction* to obtain features to characterize the detected objects.

The AROMS-Framework can be used to optimize the classification of objects inside of images, however, the three aforementioned steps have to be manually performed by experts to obtain useful feature vectors for each object. The challenge is that all these aspects interact with each other: The image preprocessing affects the segmentation process, which itself also affects the resulting object shape and texture features. Finally, the quality of these features also influences the performance of machine learning approaches. Furthermore, most image processing algorithms are also controlled by hyperparameters that need to be optimized for every task as well.

These thoughts lead to the idea of an *extended pipeline* for image-based object recognition which handles image processing and machine learning at the same time. One variant of such an extended pipeline is depicted in figure 8.1, which basically consists of the classification pipeline of the AROMS-Framework (see figure 4.2) and, additionally, image processing related pipeline elements.

The automatic and simultaneous optimization of all these pipeline elements is certainly a tremendous challenge as the configuration search space would be even larger compared to the one that was considered in this work: The image processing pipeline elements would also contain algorithm portfolios, e.g., different image preprocessing filters or segmentation methods,

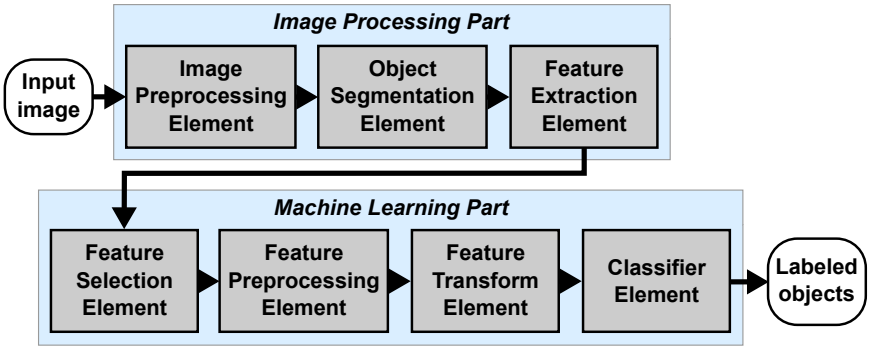


Figure 8.1: One possible variant of an extended pipeline structure to optimize image-based object segmentation and classification in a holistic way. The last four pipeline elements are the same than in the proposed classification pipeline (see figure 4.2).

with their own sets of hyperparameters. The adaptation of such an extended image processing and machine learning pipeline will most likely require a multi-objective optimization as the quality of the localization and segmentation of the objects is at least as important as the classification accuracy. Furthermore, the computational complexity of the evaluation of a configuration will rise as the image processing needs to be performed during the optimization process as well. However, it does not seem to be completely infeasible to handle this problem with Evolutionary Algorithms or model-based optimization.

Online Learning

One possible extension can be made into the direction of *online learning*. Many real-world applications need to be adapted over time to classify changing or entirely new types of instances. In this case, the training dataset has to be extended. Instead of starting a completely new optimization process, it could be helpful to use the knowledge of the previous optimization runs. One way to realize this would be to use the best configurations of the past as the initial population of the current optimization process with the extended training dataset. The hope is that this approach will speed up the optimization process when the optimal configurations only change slightly with the new dataset.

Regression

The AROMS-Framework was designed for supervised classification problems only. However, it is possible to extend it for *regression problems* as well. The main difference is that the outputs are continuous numbers instead of

discrete class labels. Basically, two modifications need to be made: At first, the classifier element has to be replaced with a pipeline element that contains regression algorithms. Secondly, a suitable objective metric for regression problems has to be used, e.g., the mean square error (MSE) to compare the actual predictions with the desired output.

8.3.4 Deep Learning Extension

The aspect of *Deep learning* has been considered in form of representation learning with feature transforms and manifold learning. Most of the current deep learning approaches are successful because of one reason: They make use of massive amounts of unlabeled training data, e.g., obtained from the internet. Most of the feature transforms that are used in the AROMS-Framework are also unsupervised. However, the current implementation requires a fully labeled training dataset. It would be helpful to be able to pass a separated set of training vectors without labels into the framework that could be used by the unsupervised representation learning algorithms. The labeled instances would still be necessary to validate the usefulness of the resulting representation.

The aspect of deep structures with many layers has turned out to be crucial for the success of deep learning. Evidence for the benefit of these deep structures is also found in the mammalian brain when visual object recognition tasks are performed. The current classification pipeline consists of four fixed pipeline elements and representation learning is only considered in one of them. It would be possible to use a more *dynamic pipeline structure* which could cascade multiple layers of processing steps that generate better representations consecutively. The optimization problem would become more complex, however, similar problems have already been successfully tackled with Evolutionary Algorithms such as Genetic Programming. However, it should be kept in mind that there is a higher risk of overfitting due to the even higher degree of adaptability.

8.3.5 Multi-Pipeline Classifier

The multi-pipeline classifier could be extended as well. One idea is to incorporate the *diversity* of the configurations into the selection of the pipeline configurations to increase the generalization performance. If the best configurations are fairly similar, the decisions of the pipelines in the multi-pipeline classifier are not independent anymore and no significant performance boost will be achieved.

The multi-pipeline classifier can also be improved by using more sophisticated aggregation functions and training methods. Some classifiers provide information about the *confidence of their decisions*, e.g., the SVM allows the

analysis of the distance of an instance to the decision boundary[3]. This information could be used to fuse only the results of the most confident classifiers into the final decision with the goal of a more reliable classification. Furthermore, the concept of *Bagging* could be introduced to train the different pipelines with generated or subsampled datasets to increase their diversity.

8.3.6 Improvement of Statistical Analyses

There is also potential to improve the *statistical analyses* to get a deeper insight into the interplay of the involved algorithms. The optimization trajectory contains even more information than the proposed multi-configuration graph reveals. The fitness values of the best configurations could also be visualized in the graph or used for an interactive exploration through the configuration distribution by selecting, e.g., the range of fitness values to be considered. In case of image-based classification problems, the frequency of two-dimensional features can be visualized as a matrix rather than a linear list. The distribution of the target dimensionality could be analyzed to get information about the intrinsic dimensionality of the problem. Furthermore, the distribution of the best hyperparameter values is interesting to identify crucial hyperparameters.

8.3.7 Summary

Table 8.1 summarizes the suggested aspects of future work with a rating regarding the improvement of the generalization performance and computational efficiency. Furthermore, the implementation effort of each suggestion is roughly estimated.

[3]The closer instances are to the decision boundary, the more risky their classification becomes.

Suggested Extension	Expected … Generalization improvement	Optimization runtime improvement	Development effort
Generalization			
– Bootstrap method	∗	–	∗
– Storage of complete models	∗	–	∗
Optimization algorithm			
– Multi-objective optimization	∗∗	∗	∗∗
– Self-adapting metaparameters	∗	∗∗	∗∗
– Model-based optimization	∗	∗	∗∗
Application field extension			
– Simult. image processing optimization	∗∗	–	∗∗
– Online learning	∗	∗	∗
– Regression	–	–	∗
Deep Learning			
– Additional unlabeled datasets	∗∗	–	∗
– Dynamic pipeline structure	∗	–	∗∗
Multi-pipeline classifier			
– Consideration of diversity	∗	–	∗
– Consideration of classifier confidence	∗	–	∗
– Combination with bagging	∗	–	∗
Statistical analyses			
– Multi-configuration graph extension	–	–	∗
– Further statistics	–	–	∗

Table 8.1: Overview of the expected impact of the suggested future extensions on the generalization performance and computational efficiency. Furthermore, the expected development effort is listed. The meaning of the scale is: ∗∗ = high, ∗ = moderate, − = not (positively) affected.

Appendix A

List of Features and Object Descriptors

This appendix lists image-based features and object descriptors that are used in the AROMS-Framework. The list is subdivided into texture and shape descriptors.

Texture Descriptors

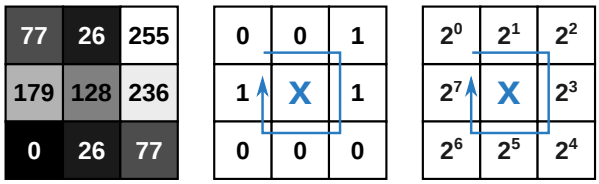
- **Color / hue channel features** are used to describe the color information of a texture. Normally, color images are acquired in the form of red-green-blue (RGB) images consisting of three separate channels. However, the color information can be extracted more robustly using the hue-saturation-value (HSV) representation of the color image [Gonzalez and Woods, 2002]. The HSV image also contains three channels, but the information is split into

  - a color angle ranging from $0 - 360°$ as the *hue* channel,
  - the saturation of the color ranging from $0 - 100$ as the *saturation* channel and
  - the brightness ranging from $0 - 100$ as the *value* channel.

  The hue channel can be used to extract simple but robust color features such as the mean and the standard deviation of the color angles.

- **Hu Moments** are gray value texture descriptors that provide invariance against translation, rotation, scaling and partly skewing of the texture. Seven features have been proposed by [Hu, 1962], which are based on central statistical image moments.

- **Local Binary Patterns (LBP)** are a family of gray value texture descriptors [Ojala et al., 2002] that are robust against monotonic brightness variations of the image. The neighbor pixels of each pixel of the



(a) Texture (b) Thresholds (c) Power of Two

Figure A.1: Principle of the basic Local Binary Patterns texture descriptor. The resulting LBP pattern for the center point is 00110001, which leads to a numeric feature value of $1 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^7 = 140$.

texture are analyzed in the following way, which is depicted in figure A.1. The center pixel is used as a gray value threshold for the pixels in the neighborhood. The resulting binary pixel values (true if a gray value is larger than the center pixel, otherwise false) are aligned as a bitstring. This bitstring is transformed into the numeric LBP value using the sum of powers of two. There are 256 possible values that can occur in the range of $[0, 255] \subset \mathbb{Z}$. In the basic LBP variant all LBP values of the texture are collected and their value distribution is represented by a histogram with 256 bins. Some variants have been introduced that provide a more robust and lower-dimensional descriptor especially for textures with only a few pixels:

  - *uniform LBP* consider only the most frequent patterns, like edges and corners, leading to 59 dimensions,
  - *rotation invariant LBP* merge all patterns that can be achieved by rotation into a single one, leading to 36 dimensions and
  - *uniform + rotation invariant LBP* combine the two aforementioned approaches and the resulting descriptor contains only 10 dimensions.

- **Low-level pixel features** are the "raw" pixel values that are scaled to a fixed image size, e.g., 20×20 pixels. The scaling usually interpolates gray values and leads to a certain degree of smoothing. The scaled image is then linearized to a vector by concatenating the image rows. The resulting vector contains the same number of elements as the number of image pixels, e.g., $20 \cdot 20 = 400$ elements. A problem is that the two-dimensional neighborhood information is lost when images are linearized in this way.

- **Scale-Invariant Feature Transform (SIFT)** is a widely used image feature detector and descriptor, introduced by [Lowe, 2004]. It is invariant to translation, scaling, rotation and partially also to distortion and illumination changes. The SIFT algorithm is usually applied

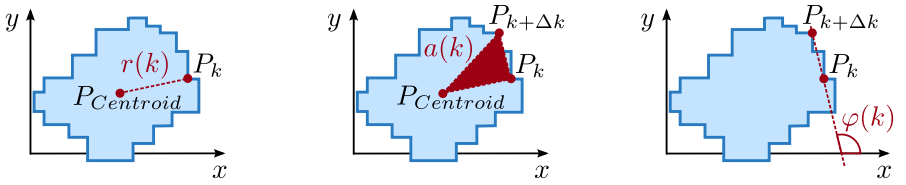
on grayscale images and basically comprises two phases, the feature detection phase and the feature description phase. At first, a set of interesting and characteristic corners is detected, which are the so-called keypoints. The second step extracts a 128 dimensional feature vector from each keypoint.

The SIFT descriptor is often applied to calculate geometrical transformations between multiple images by matching the detected keypoints between them. However, it is difficult to directly use SIFT as an object or scene descriptor in the sense of a feature vector with a fixed dimensionality: The keypoint detection and localization are inherently connected to this algorithm and the number and order of the keypoints depend on the image. Therefore, extended methods have been developed that use SIFT for a so-called bag-of-words (BoW) model [Fei-Fei and Perona, 2005], which results in a feature vector that can directly be used for classifiers.

- **Statistical histogram features** derive simple, but robust metrics from discrete distributions like value histograms with typically 256 bins (see, e.g., [Dodge, 2006] for definitions):
  - The *mean* is the average value which can also be interpreted as the center of gravity of the histogram,
  - the *standard deviation* describes the variation of the values,
  - the *skewness* measures the asymmetry of data around its mean and
  - the *kurtosis* measures the "peakedness" of the distribution.

Shape and Contour Descriptors

- **Simple shape features** are used to describe the coarse shape of segmented objects. Popular features are the following (see [Yang et al., 2008] for references):
  - The *area* describes the number of pixels in an object,
  - the *eccentricity* describes the ratio of the main axes of the object,
  - the *perimeter* describes the number of contour pixels,
  - the *circularity* describes to what extent a shape resembles a circle,
  - the *rectangularity* describes to what extent a shape resembles a rectangle and
  - the *convexity* describes the difference between the shape and its convex hull.



(a) Centroid distance $r(k)$ (b) Centroid area $a(k)$ (c) Contour angle $\varphi(k)$

Figure A.2: Visualization of the three function-based contour descriptors proposed by [Kunttu and Lepistö, 2007]. P_k denotes the kth contour point of the shape (blue object).

- **Function-based contour descriptors** are used to describe the contour of a shape more precisely. As an example there are three functions proposed by [Kunttu and Lepistö, 2007] that transform the contour into a vector:
  - The *centroid distance* function describes the variation of the distance to the center along the contour,
  - the *centroid area* function describes the variation of the triangle area that is formed by two contour points and the center point along the contour and
  - the *contour angle* function describes the variation of the tangential angle at the contour point along the contour.

  The final feature vector is obtained by the Fourier transform of this vector to achieve a shape size invariance and to remove noise. Figure A.2 illustrates the three descriptors with an example shape.

Appendix B

List of Feature Preprocessing Methods

This appendix lists the feature preprocessing methods in the algorithm portfolio $S_{PreProc}$, which is used in the classification pipeline of the AROMS-Framework.

Most of the feature preprocessing methods estimate model parameters of the feature distribution of the training dataset T_{Train} using the concatenated feature vector matrix of all training vectors

$$\mathbf{X}_T = \begin{bmatrix} \mathbf{x}^{(1)} \\ \mathbf{x}^{(2)} \\ \vdots \\ \mathbf{x}^{(N_T)} \end{bmatrix} \in \mathbb{R}^{N_T \times D_{in}} \ . \tag{B.1}$$

A column of this matrix

$$\mathbf{x}_{T,j} = \left[x_j^{(1)}, x_j^{(2)}, \ldots, x_j^{(N_T)}\right]^\top \tag{B.2}$$

contains all feature values of the jth dimension within the training dataset T_{Train}.

Preprocessing Methods

- **Rescaling** scales each feature dimension to a specific range so that all values in $\mathbf{x}_{T,j}$ lie in the range of $[0, 1] \subset \mathbb{R}$. This is a well known preprocessing step to improve the classification performance [Juszczak et al., 2002]. It is achieved by calculating the minimum and maximum values in $\mathbf{x}_{T,j}$ for all dimensions. The actual rescaling of a new vector is done separately for each dimension and the jth dimension x_j is transformed by

$$x_{rescaled,j} = \frac{x_j - \min(\mathbf{x}_{T,j})}{\max(\mathbf{x}_{T,j}) - \min(\mathbf{x}_{T,j})} \tag{B.3}$$

for $\max(\mathbf{x}_{T,j}) - \min(\mathbf{x}_{T,j}) > 0$. This is calculated for all dimensions $1 \leq j \leq D_{in}$.

- **Standardization** is similar to rescaling, however, the data is scaled to a normal distribution with zero mean and a standard deviation value of one. The estimation of these model parameters is usually more robust than considering only the minimum and maximum values. The standardization of the jth dimension x_j is done by

$$x_{standard,j} = \frac{x_j - \text{mean}(\mathbf{x}_{T,j})}{\text{std}(\mathbf{x}_{T,j})} \tag{B.4}$$

for $\text{std}(\mathbf{x}_{T,j}) > 0$. This is calculated for all dimensions $1 \leq j \leq D_{in}$.

- **L2-Normalization** scales each feature vector $\mathbf{x}^{(i)}$ independently from the training dataset T_{Train} to unit length, which is equivalent to an L_2-norm value of one:

$$\mathbf{x}_{norm} = \frac{\mathbf{x}}{\|\mathbf{x}\|_2} \ , \quad \|\mathbf{x}\|_2 > 0 \ . \tag{B.5}$$

- **Pre-Whitening** is a more complex preprocessing step that performs a decorrelation transformation resulting to a feature matrix $\mathbf{X}_T$ with zero mean and having a covariance matrix equal to the identity matrix [Bishop, 2006]. It helps to remove redundancies within features that is often a problem of low-level image-based data in which neighboring pixels tend to be highly correlated.
- **Identity / no preprocessing** is part of the portfolio as well because there may be datasets for which any other preprocessing step is counterproductive.

Appendix C

List of Feature Transforms

This appendix lists the feature transforms and manifold learning algorithms of the algorithm portfolio S_{Trans}, which is used in the classification pipeline of the AROMS-Framework. All these algorithms fit to the proposed feature transform interface (see chapter 3.3.2) and are implemented in the *Matlab Toolbox for Dimensionality Reduction* [Van der Maaten, 2014]. In total, there are $N_{Trans} = 31$ unique feature transforms in this algorithm portfolio.

This list provides important information about the feature transforms, namely their name and abbreviation, a brief description, a reference and three properties (linear or non-linear, supervised or non-supervised and the type of out-of-sample extension). Furthermore, the hyperparameters of the feature transforms are listed. Note that some methods adapt other hyperparameters automatically. Most of the references are taken from [Ma and Fu, 2011] and [Van der Maaten et al., 2009], which can also be used for further information.

The list is provided in alphabetical order and some feature transforms are marked as follows:

* This transform is not used within the experiments due to duplicate functionality with related transforms.

** An out-of-sample extension exists but it is not available in the Matlab Toolbox for Dimensionality Reduction.

- **Autoencoder**: Artificial neural network that reconstructs the input through a bottleneck layer [Hinton and Salakhutdinov, 2006].

  Properties: non-linear, unsupervised, direct parametric out-of-sample extension

  Hyperparameters:
  - regularization parameter: real-valued, values $\in [0, 1] \subset \mathbb{R}$, default=0

- **CFA (Coordinated Factor Analysis)**: Combination of locally linear mappings for a global, non-linear mapping [Verbeek, 2006].

  Properties: non-linear, unsupervised, no out-of-sample extension

  Hyperparameters:
  - number of analyzers: integer, values $\in [1, 100] \subset \mathbb{Z}$, default=40
  - maximum number of iterations: integer, values $\in [1, 500] \subset \mathbb{Z}$, default=200

- **Diffusion Maps**: Distance metric based on Markov random walks on a graph representation of the data [Lafon and Lee, 2006].

  Properties: non-linear, unsupervised, approximation for out-of-sample extension **

  Hyperparameters:
  - number of timesteps: integer, values $\in [1, 5] \subset \mathbb{Z}$, default=1
  - Gaussian kernel γ_{Gauss}: exponentially scaled, real-valued, values $\in [10^{-5}, 10^{2}] \subset \mathbb{R}$, default=1

- **Factor Analysis**: Analysis of the underlying factors that describe intercorrelations between variables [Spearman, 1904].

  Properties: linear, unsupervised, direct parametric out-of-sample extension

  Hyperparameters: –

- **FastICA**: A fast implementation of ICA based on an iterative algorithm [Hyvärinen, 1999].

  Properties: linear, unsupervised, direct parametric out-of-sample extension

  Hyperparameters: –

- **GPLVM (Gaussian Process Latent Variable Models)**: Non-linear variant of the probabilistic Kernel-PCA [Lawrence, 2005].

  Properties: non-linear, unsupervised, no out-of-sample extension

  Hyperparameters:
  - Gaussian kernel γ_{Gauss}: exponentially scaled, real-valued, values $\in [10^{-5}, 10^{2}] \subset \mathbb{R}$, default=1

- **Hessian LLE**: Extension of LLE based on the Hessian matrix to minimize the curviness of the manifold [Donoho and Grimes, 2003].

  Properties: non-linear, unsupervised, no out-of-sample extension

  Hyperparameters:
  - number of nearest neighbors: integer, values $\in [1, 20] \subset \mathbb{Z}$, default=12

- **ICA (Independent Component Analysis*)**: Analysis of independent signals or components with an additive signal model (the FastICA implementation is used) [Hyvärinen and Oja, 2000].

  Properties: linear, unsupervised, direct parametric out-of-sample extension

  Hyperparameters: –

- **Identity (No transform)**: The features are not changed and no dimensionality reduction is done.

  Properties: "linear", "unsupervised", "direct parametric out-of-sample extension"

  Hyperparameters: –

- **Isomap**: Nearest neighbor graph-based estimation of the geodesic distance [Tenenbaum et al., 2000].

  Properties: non-linear, unsupervised, approximation for out-of-sample extension

  Hyperparameters:
  - number of nearest neighbors: integer, values $\in [1, 20] \subset \mathbb{Z}$, default=12

- **Kernel-LDA (Extension to LDA with, e.g., Gaussian or polynomial kernels)**: Kernel extension to LDA [Mika et al., 1999].

  Properties: non-linear, supervised, no out-of-sample extension

  Hyperparameters:
  - Gaussian kernel: γ_{Gauss}: exponentially scaled, real-valued, values $\in [10^{-5}, 10^{2}] \subset \mathbb{R}$, default=1
  - polynomial kernel: δ: real-valued, values $\in [1, 3] \subset \mathbb{R}$, default=3
  - polynomial kernel: η: integer, values $\in [0, 10] \subset \mathbb{Z}$, default=1

- **Kernel-PCA (Extension to PCA with, e.g., Gaussian or polynomial kernels)**: Kernel extension to the PCA on distance matrices [Schölkopf et al., 1998].

  Properties: non-linear, unsupervised, direct parametric out-of-sample extension

  Hyperparameters:
  - Gaussian kernel: γ_{Gauss}: exponentially scaled, real-valued, values $\in [10^{-5}, 10^{2}] \subset \mathbb{R}$, default=1
  - polynomial kernel: δ: real-valued, values $\in [1, 3] \subset \mathbb{R}$, default=3
  - polynomial kernel: η: integer, values $\in [0, 10] \subset \mathbb{Z}$, default=1

- **Landmark Isomap**: More efficient version of Isomap using a subset of feature vectors as landmarks [Silva and Tenenbaum, 2002].

  Properties: non-linear, unsupervised, approximation for out-of-sample extension

  Hyperparameters:
  - number of nearest neighbors: integer, values $\in [1, 20] \subset \mathbb{Z}$, default=12
  - percentage of landmark points: real-valued, values $\in [0, 1] \subset \mathbb{R}$, default=0.2

- **Laplacian Eigenmaps**: Similar to LLE and preserves the local distances [Belkin and Niyogi, 2001].

  Properties: non-linear, unsupervised, approximation for out-of-sample extension

  Hyperparameters:
  - number of nearest neighbors: integer, values $\in [1, 20] \subset \mathbb{Z}$, default=12
  - Gaussian kernel γ_{Gauss}: exponentially scaled, real-valued, values $\in [10^{-5}, 10^{2}] \subset \mathbb{R}$, default=1

- **LDA (Linear Discriminant Analysis/Fisher Discriminant Analysis/FDA)**: Linear projection to optimize the class separability, target dimensionality is fixed to the number of classes -1 [Fisher, 1936].

  Properties: linear, supervised, direct parametric out-of-sample extension

  Hyperparameters: –

- **LLC (Locally Linear Coordination)**: Global alignment of locally linear models [Teh and Roweis, 2002].

  Properties: non-linear, unsupervised, no out-of-sample extension

  Hyperparameters:
  - number of nearest neighbors: integer, values $\in [1, 20] \subset \mathbb{Z}$, default=12
  - number of analyzers: integer, values $\in [1, 100] \subset \mathbb{Z}$, default=20
  - maximum number of iterations: integer, values $\in [50, 500] \subset \mathbb{Z}$, default=200

- **LLE (Locally Linear Embedding)**: Reconstruction of distances to the nearest neighbors [Roweis and Saul, 2000].

  Properties: non-linear, unsupervised, approximation for out-of-sample extension

  Hyperparameters:
  - number of nearest neighbors: integer, values $\in [1, 20] \subset \mathbb{Z}$, default=12

- **LLTSA (Linear LTSA)**: Linear extension for LTSA [Zhang et al., 2007].

  Properties: linear, unsupervised, direct parametric out-of-sample extension

  Hyperparameters:
  - number of nearest neighbors: integer, values $\in [1, 20] \subset \mathbb{Z}$, default=12

- **LMNN (Large-Margin Nearest Neighbor)**: Learn a Mahalanobis distance to achieve a linear transform with a large margin between classes, target dimensionality is fixed to the input dimensionality [Weinberger and Saul, 2009].

  Properties: linear, supervised, direct parametric out-of-sample extension

  Hyperparameters:
  - number of nearest neighbors: integer, values $\in [1, 20] \subset \mathbb{Z}$, default=3

- **LPP (Locality Preserving Projection)**: Linear transform to optimally preserve the neighborhood structure [Niyogi, 2004].

  Properties: linear, unsupervised, direct parametric out-of-sample extension

  Hyperparameters:
  - number of nearest neighbors: integer, values $\in [1, 20] \subset \mathbb{Z}$, default=12
  - Gaussian kernel γ_{Gauss}: exponentially scaled, real-valued, values $\in [10^{-5}, 10^{2}] \subset \mathbb{R}$, default=1

- **LTSA (Local Tangent Space Analysis)**: Computing a mapping into locally linear tangents by applying a local PCA [Zhang and Zha, 2004].

  Properties: non-linear, unsupervised, no out-of-sample extension

  Hyperparameters:
  - number of nearest neighbors: integer, values $\in [1, 20] \subset \mathbb{Z}$, default=12

- **Manifold Charting**: Similar to LLC with a different cost function [Brand, 2002].

  Properties: non-linear, unsupervised, approximation for out-of-sample extension

  Hyperparameters:
  - number of analyzers: integer, values $\in [1, 100] \subset \mathbb{Z}$, default=40
  - maximum number of iterations: integer, values $\in [50, 500] \subset \mathbb{Z}$, default=200

- **MCML (Maximally Collapsing Metric Learning)**: Linear transform that projects instances of the same class into the same location of the target space, similar to NCA [Globerson and Roweis, 2005].

  Properties: linear, supervised, direct parametric out-of-sample extension

  Hyperparameters: –

- **MDS (Multidimensional/Classical Scaling*)**: Dimension reduction framework relying on pairwise distance matrices between instances (the simplest variant is equivalent to PCA which is actually used in the framework) [Torgerson, 1952].

  Properties: linear or non-linear, unsupervised, approximation for out-of-sample extension**

  Hyperparameters: –

- **NCA (Neighborhood Components Analysis)**: Linear transform based on a distance metric that optimizes the nearest neighbor classification [Goldberger et al., 2004].

  Properties: linear, supervised, direct parametric out-of-sample extension

  Hyperparameters:

  - regularization parameter: real-valued, values $\in [0, 1] \subset \mathbb{R}$, default=0

- **NPE (Neighborhood Preserving Embedding)**: Linear transform to preserve the local manifold structure [He et al., 2005].

  Properties: linear, unsupervised, direct parametric out-of-sample extension

  Hyperparameters:

  - number of nearest neighbors: integer, values $\in [1, 20] \subset \mathbb{Z}$, default=12

- **PCA (Principal Component Analysis)**: Linear transform based on the statistical projection on the axis of the highest variance [Pearson, 1901].

  Properties: linear, unsupervised, direct parametric out-of-sample extension

  Hyperparameters: –

- **Sammon Mapping**: Similar to MDS, but retains small pairwise distances in the lower-dimensional space [Sammon, 1969].

  Properties: non-linear, unsupervised, no out-of-sample extension

  Hyperparameters: –

- **SNE (Stochastic Neighbor Embedding)**: Asymmetric neighborhood probability model with a Gaussian distribution [Hinton and Roweis, 2002].

  Properties: non-linear, unsupervised, no out-of-sample extension

  Hyperparameters:

  - regularization parameter: integer, values $\in [0, 100] \subset \mathbb{Z}$, default=30

- **SPE (Structure Preserving Embedding)**: Algorithm to compress graphs that preserves the global topology [Shaw and Jebara, 2009].

  Properties: non-linear, unsupervised, no out-of-sample extension

  Hyperparameters:
  - variant: categorical, values $\in \{Global, Local\}$, default=*Global*
  - number of nearest neighbors: integer, values $\in [1, 20] \subset \mathbb{Z}$, default=12

- **Symmetric SNE**: Like SNE, but with a symmetric probability model [Van der Maaten and Hinton, 2008].

  Properties: non-linear, unsupervised, no out-of-sample extension

  Hyperparameters:
  - regularization parameter: integer, values $\in [0, 100] \subset \mathbb{Z}$, default=30

- **t-SNE (parametric)**: A deep neural network that minimizes a similar cost function as t-SNE [Van der Maaten, 2009].

  Properties: non-linear, unsupervised, direct parametric out-of-sample extension

  Hyperparameters:
  - number neurons on layer 1: integer, values $\in [1, 500] \subset \mathbb{Z}$, default=200
  - number neurons on layer 2: integer, values $\in [1, 500] \subset \mathbb{Z}$, default=200
  - number neurons on layer 3: integer, values $\in [1, 2000] \subset \mathbb{Z}$, default=1000

- **t-SNE**: Like SNE, but with a Student t-probability distribution [Van der Maaten and Hinton, 2008].

  Properties: non-linear, unsupervised, no out-of-sample extension

  Hyperparameters:
  - regularization parameter: integer, values $\in [0, 100] \subset \mathbb{Z}$, default=30
  - number of initial dimensions: integer, values $\in [0, 50] \subset \mathbb{Z}$, default=30

Appendix D

List of Classifiers

This appendix lists the classifiers in the algorithm portfolio $S_{Classifiers}$, which is used in the classification pipeline of the AROMS-Framework. Furthermore, the corresponding hyperparameters and value domains are listed.

- **Extreme Learning Machine** (ELM): Artificial neural network with one hidden layer and random weights [Huang et al., 2006]; the original implementation of [Huang et al., 2006] is used. Hyperparameters:

  (i) *number of neurons on hidden layer*, type integer
  - value range: $[1, 1000] \subset \mathbb{Z}$
  - default value: 100

- **Multilayer Perceptron** (MLP): Artificial neural network with multiple hidden layers, which is trained with a backpropagation algorithm [Bishop, 2006]; the implementation of *Matlab Neural Network Toolbox* is used. Hyperparameters:

  (i) *number of hidden layers*, type integer
  - value range: $[0, 3] \subset \mathbb{Z}$
  - default value: 2

  (ii) *number of neurons on each hidden layer*, type integer
  - value range: $[1, 30] \subset \mathbb{Z}$
  - default value: 5

- **Naïve Bayes classifier**: Simple and fast statistical classifier that estimates the class probabilities independently for each feature using a Gaussian probability model [Bishop, 2006]; the implementation of *Matlab Statistics Toolbox* is used. Hyperparameters: none.

- **Nearest Neighbors classifier** (kNN): Instance-based classifier considering the N_{Neigh} nearest neighbors to a given sample according to a specified distance metric [Cover and Hart, 1967]; the implementation of *Matlab Statistics Toolbox* is used. Hyperparameters:

  (i) N_{Neigh}: number of neighbors, type integer
  - value range: $[1, 20] \subset \mathbb{Z}$
  - default value: 3

  (ii) *distance metric*: distance function for the nearest neighbor determination, type categorical
  - values: {*Euclidean*, *Mahalanobis*, *Cityblock*, *Chebychev*}
  - default value: *Euclidean*

- **Random Forest** (RF): Combination of multiple decision tree classifiers [Breiman, 2001]; the implementation of *Matlab Statistics Toolbox* is used. Hyperparameters:

  (i) *number of trees*, type integer
  - value range: $[10, 500] \subset \mathbb{Z}$
  - default value: 50
  - grid-based tuning for baseline classifier: $\{10, 50, 200\}$

- **Support Vector Machine with a linear kernel** (SVM linear): Popular classifier which maximizes the margin around the decision boundary between classes [Burges, 1998]; the multiclass-capable implementation of LIBSVM [Chang and Lin, 2011] is used. Hyperparameters:

  (i) c_{reg}: regularization parameter for the cost of misclassifications, type exponentially scaled, real-valued
  - value range: $[10^{-2}, 10^{4}] \subset \mathbb{R}$
  - default value: 1

- **Support Vector Machine with a Gaussian kernel** (SVM Gauss): Gaussian kernel extension of the linear SVM (see above). Hyperparameters:

  (i) c_{reg}: regularization parameter for the cost of misclassifications, type exponentially scaled, real-valued
  - value range: $[10^{-2}, 10^{4}] \subset \mathbb{R}$
  - default value: 1
  - grid-based tuning for baseline classifier: $\{10^{-2}, 10^{0}, 10^{2}\}$

  (ii) γ_{Gauss}: Gaussian kernel width, type exponentially scaled, real-valued
  - value range: $[10^{-5}, 10^{2}] \subset \mathbb{R}$
  - default value: 10^{-1}
  - grid-based tuning for baseline classifier: $\{10^{-4}, 10^{-1}, 10^{2}\}$

- **Support Vector Machine with a polynomial kernel** (SVM Poly): Polynomial kernel extension of the SVM (see above). Hyperparameters:

  (i) c_{reg}: regularization parameter for the cost of misclassifications, type exponentially scaled, real-valued
  - value range: $[10^{-2}, 10^{4}] \subset \mathbb{R}$
  - default value: 1

  (ii) η: polynomial degree, type integer
  - value range: $[2, 5] \subset \mathbb{Z}$
  - default value: 3

Appendix E

Metaparameters of the AROMS-Framework

This appendix lists metaparameters of the AROMS-Framework and their default values in table E.1 and E.2.

Metaparameter	*Type*	*Value*	*Description*
$\Delta_{fit,max}$	$\mathbb{R}$	0.03	Fitness threshold for the multi-pipeline classifier (see section 6.3.3)
Δ_t	$\mathbb{N}$	3	Generation offset for the termination criterion (see sections 5.4.2 and 5.5.3)
ϵ_{fit}	$\mathbb{R}$	0.001	Minimal fitness improvement for the termination criterion (see sections 5.4.2 and 5.5.3)
κ	$\mathbb{N}$	4	Maximal individual lifespan in number of generations (see sections 5.4.2 and 5.5.3)
λ	$\mathbb{N}$	200	Number of individuals generated in each generation (see sections 5.4.2 and 5.5.3)
μ	$\mathbb{N}$	20	Number of individuals that survive each generation (see sections 5.4.2 and 5.5.3)
μ_{init}	$\mathbb{N}$	400	Number of initial individuals in a population (see sections 5.4.2 and 5.5.3)
ρ	$\mathbb{N}$	3	Number of parents (see sections 5.4.2 and 5.5.3)
τ_1	$\mathbb{R}$	0.5	Weight for the global mutation adaptation (see section 5.4.2)
τ_2	$\mathbb{R}$	0.5	Weight for the local mutation adaptation (see section 5.4.2)
χ_{Mut}	$\mathbb{R}$	0.2	Initial mutation ratio of the value domain of numerical variables (see section 5.4.2)

Table E.1: Metaparameters of the AROMS-Framework Part I.

Metaparameter	*Type*	*Value*	*Description*
$D_{Trans,Max}$	$\mathbb{N}$	50	Maximum target dimensionality (see section 5.5.1)
earlyDiscarding	$\mathbb{B}$	true	Early discarding system (see section 5.1.3)
ECA-variant	categ.	*ECA-full*	Determination of the variant of the *ECA* algorithm (see section 5.6)
fit_{high}	$\mathbb{R}$	1	Upper fitness bound for roulette wheel selection (see section 5.4.2)
fit_{low}	$\mathbb{R}$	0.25	Lower fitness bound for roulette wheel selection (see section 5.4.2)
holisticCVactive	$\mathbb{B}$	true	Holistic cross-validation (see section 5.1.2)
initPopImpr	$\mathbb{B}$	true	Improvement of the initial population by random forest variable importance (see section 5.5.4)
$N_{Configs}$	$\mathbb{N}$	50	Number of configurations for the multi-configuration graph (see section 6.2.1)
N_{CV}	$\mathbb{N}$	5	Number of cross-validation rounds (see section 5.1.2)
$N_{FeatGroupMax}$	$\mathbb{N}$	16	Maximum number of feature groups for the multi-configuration graph (see section 6.2.1)
$N_{Generations,min}$	$\mathbb{N}$	5	Minimum number of generations (see sections 5.4.2 and 5.5.3)
N_{Pipes}	$\mathbb{N}$	25	Number of pipelines for the multi-pipeline classifier (static selection) (see section 6.3.3)
$N_{Pipes,max}$	$\mathbb{N}$	50	Maximum number of pipelines for the multi-pipeline classifier (see section 6.3.3)
$p_{Feat,min}$	$\mathbb{R}$	0.25	Minimum initialization probability for feature selection variables (see section 5.5.4)
$p_{Init,\mathbb{B}}$	$\mathbb{R}$	0.5	Initial probability for Boolean variables in case no other initialization is used (see section 5.4.2)
$p_{Mut,\mathbb{B},init}$	$\mathbb{R}$	0.1	Initial mutation probability for Boolean variables (see section 5.4.2)
$p_{Mut,\mathbb{S},init}$	$\mathbb{R}$	0.2	Initial mutation probability for categorical variables (see section 5.4.2)
$S_{Classifiers}$	set	all	Portfolio of classifiers (see section 4.5.4 and appendix D for all methods)
$S_{PreProc}$	set	all	Portfolio of feature preprocessing methods (see section 4.5.2 and appendix B for all methods)
S_{Trans}	set	all	Portfolio of feature transforms (see section 4.5.3 and appendix C for all methods)

Table E.2: Metaparameters of the AROMS-Framework Part II.

Appendix F

Statistical Test Methods

This appendix describes statistical test methods that are used in chapters 5 and 7. For a deeper insight into the field of statistics, the reader is kindly referred to books such as [Howell, 2006] or [Field, 2013].

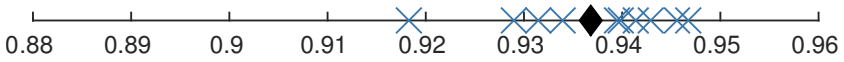
Motivation for Statistical Tests

The proposed Evolutionary Configuration Adaptation (*ECA*) optimization algorithm produces a number of outputs and metrics that can be compared, e.g., the cross-validation accuracy, the generalization accuracy or the optimization runtime. The *ECA* algorithm is based on Evolutionary Algorithms that rely on random processes and thus not all results are deterministic. Therefore, the experiments have to be repeated multiple times. In order to compare the results of multiple algorithm variants on the same dataset the distributions of the target metrics have to be compared.

Figure F.1 shows the cross-validation accuracy results of three (imaginary) algorithms on the same dataset while each experiment has been repeated ten times. The question is which of the three algorithms is the best regarding the cross-validation accuracy. The most obvious approach is to simply compare the means of the three cross-validation accuracy distributions. The mean of algorithm A_1 is higher than the means of the algorithms A_2 and A_3 while the means of A_2 and A_3 are very similar. A first incautious conclusion would be to assume that algorithm A_1 is better than A_2 and A_3. However, it is not obvious if the differences between the algorithms are *significant*. If the differences are not significant, there is the risk that a higher number of repetitions could change the comparative results.

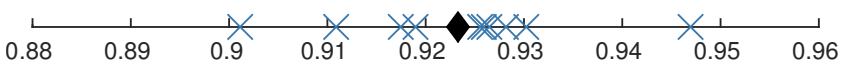
Suitable Tests to Compare two Means

Statistical tests are required in order to find out if the means of two independent value distributions differ significantly from each other or not.



(a) Algorithm A_1: mean = 0.9369, standard deviation = 0.0088

(b) Algorithm A_2: mean=0.9229, standard deviation = 0.0194



(c) Algorithm A_3: mean=0.9232, standard deviation = 0.0123

Figure F.1: Imaginary cross-validation accuracy distributions of three algorithms. The means are indicated with the black diamond symbol and the single results with cross symbols.

Let $\mathbf{x}^{(1)} \in \mathbb{R}^{D_1}$ and $\mathbf{x}^{(2)} \in \mathbb{R}^{D_2}$ be two vectors with the metrics that should be compared. The corresponding means are calculated using

$$\overline{x}^{(1)} = \frac{1}{D_1} \sum_{i=1}^{D_1} x_i^{(1)} \quad \text{and} \quad \overline{x}^{(2)} = \frac{1}{D_2} \sum_{i=1}^{D_2} x_i^{(2)} \tag{F.1}$$

in which x_i denotes the ith vector component. The standard deviations

$$\sigma_1 = \sqrt{\frac{1}{D_1 - 1} \sum_{i=1}^{D_1} (x_i^{(1)} - \overline{x}^{(1)})^2} \quad \text{and} \quad \sigma_2 = \sqrt{\frac{1}{D_2 - 1} \sum_{i=1}^{D_2} (x_i^{(2)} - \overline{x}^{(2)})^2} \tag{F.2}$$

are also calculated. The selection of a suitable statistical test method depends on several properties of the data. The sample sizes D_1 and D_2 are relatively low in the experiments, e.g., ten repetitions are used and so ten values are available for each algorithm. The *Student's t-test* is suitable to compare means of two distributions [Student, 1908], however, it assumes that the standard deviations σ_1 and σ_2 are equal. Nevertheless, this cannot be assumed when results of different *ECA* variants are compared as some variants might terminate in local optima more often – which would directly lead to a higher standard deviation of the metrics. The t-test has been extended by [Welch, 1947] to work for the case $\sigma_1 \neq \sigma_2$ as well, which is known as *Welch's unequal variances t-test*, or simply *Welch test*. Consequently, the Welch test needs to be used to compare the results of different algorithms in the evaluations.

The Welch Test

The Welch test is a two-sample test with the null hypothesis that the two means $\overline{x}^{(1)}$ and $\overline{x}^{(2)}$ are equal. The assumptions are that $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ are independent and normally distributed and that the standard deviations are not necessarily equal. The sample sizes D_1 and D_2 may differ as well. To

perform the test, a level of significance, typically $\alpha_{Welch} = 0.05$, is chosen which determines the risk of rejecting the null hypothesis even though it is true – this is also called the type I error. The results of the test is the p_{Welch}-value that is proportional to the risk of committing a type I error. The smaller p_{Welch} is, the less risky it becomes to reject the null hypothesis. If it is rejected, it is assumed that $\overline{x}^{(1)} \neq \overline{x}^{(2)}$.

The Welch test leads to the following results when it is applied to the data in figure F.1:

- The cross-validation accuracy results of algorithm A_1 and A_2 are *not* significantly different for $\alpha_{Welch} = 0.05$ ($p_{Welch} = 0.0591$), even though it is a border case. However, the statement "algorithm A_1 is better than A_2" is not legitimate based on the available data.
- The cross-validation accuracy results of algorithm A_1 and A_3 are significantly different for $\alpha_{Welch} = 0.05$ ($p_{Welch} = 0.0112$). The statement "algorithm A_1 is better than A_3" is statistically supported by the data.

These results might be surprising because the means of A_2 and A_3 are almost identical. However, the larger standard deviation of the cross-validation accuracy values of algorithm A_2 explain the result of the Welch test.

List of Figures

List of Tables

Bibliography

[Åberg and Wessberg, 2007] Åberg, M. and Wessberg, J. (2007). Evolutionary optimization of classifiers and features for single trial EEG discrimination. *Biomedical engineering online*, 6(1):32.

[Aha, 1997] Aha, D. W. (1997). *Lazy learning*. Kluwer academic publishers.

[Aha et al., 1991] Aha, D. W., Kibler, D., and Albert, M. K. (1991). Instance-based learning algorithms. *Machine learning*, 6(1):37–66.

[Albanese et al., 2012] Albanese, D., Visintainer, R., Merler, S., Riccadonna, S., Jurman, G., and Furlanello, C. (2012). mlpy: Machine Learning Python.

[Ansótegui et al., 2009] Ansótegui, C., Sellmann, M., and Tierney, K. (2009). A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms. In Gent, I., editor, *Principles and Practice of Constraint Programming - CP 2009*, volume 5732 of *Lecture Notes in Computer Science*, pages 142–157. Springer Berlin Heidelberg.

[Bache and Lichman, 2013] Bache, K. and Lichman, M. (2013). UCI Machine Learning Repository, `http://archive.ics.uci.edu/ml` [accessed 28.09.2015 12:00].

[Bäck, 1996] Bäck, T. (1996). *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, Oxford, UK.

[Belkin and Niyogi, 2001] Belkin, M. and Niyogi, P. (2001). Laplacian eigenmaps and spectral techniques for embedding and clustering. In *Advances in Neural Information Processing Systems (NIPS)*, volume 14, pages 585–591.

[Bellman, 1961] Bellman, R. E. (1961). *Adaptive control processes: a guided tour*, volume 4. Princeton University Press Princeton.

[Bengio et al., 2013] Bengio, Y., Courville, A., and Vincent, P. (2013). Representation learning: A review and new perspectives. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35(8):1798–1828.

[Bengio et al., 2004] Bengio, Y., Paiement, J.-F., Vincent, P., Delalleau, O., Le Roux, N., and Ouimet, M. (2004). Out-of-sample extensions for LLE, Isomap, MDS, eigenmaps, and spectral clustering. *Advances in neural information processing systems*, 16:177–184.

[Bergstra et al., 2011] Bergstra, J., Bardenet, R., Bengio, Y., Kégl, B., et al. (2011). Algorithms for hyper-parameter optimization. In *25th Annual Conference on Neural Information Processing Systems (NIPS 2011)*.

[Bergstra and Bengio, 2012] Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13(1):281–305.

[Beyer and Schwefel, 2002] Beyer, H.-G. and Schwefel, H.-P. (2002). Evolution strategies–A comprehensive introduction. *Natural computing*, 1(1):3–52.

[Beyer et al., 1999] Beyer, K., Goldstein, J., Ramakrishnan, R., and Shaft, U. (1999). When Is "Nearest Neighbor" Meaningful? In Beeri, C. and Buneman, P., editors, *Database Theory ICDT 1999*, volume 1540 of *Lecture Notes in Computer Science*, pages 217–235. Springer Berlin Heidelberg.

[Bishop, 1995] Bishop, C. (1995). *Neural networks for pattern recognition*. Clarendon Press Oxford University Press, Oxford New York.

[Bishop, 2006] Bishop, C. M. (2006). *Pattern recognition and machine learning*, volume 1. Springer New York.

[Blum and Roli, 2003] Blum, C. and Roli, A. (2003). Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys (CSUR)*, 35(3):268–308.

[Booch et al., 2005] Booch, G., Rumbaugh, J., and Jacobson, I. (2005). *The Unified Modeling Language User Guide (2nd Edition)*. Addison-Wesley Professional.

[Brand, 2002] Brand, M. (2002). Charting a manifold. In *Advances in neural information processing systems*, pages 961–968. MIT Press.

[Breiman, 1996] Breiman, L. (1996). Bagging predictors. *Machine Learning*, 24(2):123–140.

[Breiman, 2001] Breiman, L. (2001). Random Forests. *Machine Learning*, 45(1):5–32.

[Brochu et al., 2010] Brochu, E., Cora, V. M., and De Freitas, N. (2010). A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*.

[Buck et al., 2013] Buck, C., Bürger, F., Herwig, J., and Thurau, M. (2013). Rapid Inclusion and Defect Detection System for Large Steel Volumes. *ISIJ International*, 53(11):1927–1935.

[Bürger, 2016] Bürger, F. (2016). Source code publication of the Matlab implementation of the AROMS-Framework on Github, `https://github.com/FabianBuerger/AROMS-MachineLearning` [accessed 11.1.2016 12:00].

[Bürger et al., 2014] Bürger, F., Buck, C., Pauli, J., and Luther, W. (2014). Image-based Object Classification of Defects in Steel using Data-driven Machine Learning Optimization. In Braz, J. and Battiato, S., editors, *Proceedings of VISAPP 2014 - International Conference on Computer Vision Theory and Applications*, volume 2, pages 143–152. SCITEPRESS, Lisbon, Portugal.

[Bürger et al., 2013] Bürger, F., Herwig, J., Thurau, M., Buck, C., Luther, W., and Pauli, J. (2013). An Auto-adaptive Measurement System for Statistical Modeling of Non-metallic Inclusions through Image-based Analysis of Milled Steel Surfaces. In Bosse, H. and Schmitt, R., editors, *ISMTII 2013, 11th International Symposium on Measurement Technology and Intelligent Instruments*. Apprimus Wissenschaftsverlag.

[Bürger and Pauli, 2013] Bürger, F. and Pauli, J. (2013). Unsupervised Segmentation of Anomalies in Sequential Data, Images and Volumetric Data Using Multiscale Fourier Phase-Only Analysis. In Kämäräinen, J.-K. and Koskela, M., editors, *Image Analysis - 18th Scandinavian Conference, SCIA 2013, Espoo, Finland, June 17-20, 2013, Proceedings*, volume 7944 of *Lecture Notes in Computer Science*, pages 44–53. Springer Berlin Heidelberg.

[Bürger and Pauli, 2015a] Bürger, F. and Pauli, J. (2015a). Automatic Representation and Classifier Optimization for Image-based Object Recognition. In Battiato, S. and Imai, F., editors, *Proceedings of the 10th International Conference on Computer Vision Theory and Applications (VISAPP 2015)*, volume 2, pages 542–550. SCITEPRESS, Lisbon, Portugal.

[Bürger and Pauli, 2015b] Bürger, F. and Pauli, J. (2015b). A Holistic Classification Optimization Framework with Feature Selection, Preprocessing, Manifold Learning and Classifiers. In Fred, A., Marsico, M. D., and Figueiredo, M., editors, *Pattern Recognition: Applications and Methods*

- 4th International Conference, ICPRAM 2015, Lisbon, Portugal, January 10-12, 2015, Revised Selected Papers, volume 9493 of *Lecture Notes in Computer Science*, pages 52–68. Springer International Publishing.

[Bürger and Pauli, 2015c] Bürger, F. and Pauli, J. (2015c). Representation Optimization with Feature Selection and Manifold Learning in a Holistic Classification Framework. In De Marsico, M. and Fred, A., editors, *ICPRAM 2015 - Proceedings of the International Conference on Pattern Recognition Applications and Methods*, volume 1, pages 35–44. SCITEPRESS, Lisbon, Portugal.

[Bürger and Pauli, 2016] Bürger, F. and Pauli, J. (2016). Understanding the Interplay of Simultaneous Model Selection and Representation Optimization for Classification Tasks. In De Marsico, M., Sanniti di Baja, G., and Fred, A., editors, *ICPRAM 2016 - Proceedings of the 5th International Conference on Pattern Recognition Applications and Methods*, volume 1, pages 283–290. SCITEPRESS, Lisbon, Portugal.

[Burges, 1998] Burges, C. (1998). A Tutorial on Support Vector Machines for Pattern Recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167.

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., and Stal, M. (1996). *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley.

[Carbonell, 1983] Carbonell, J. G. (1983). Learning by Analogy: Formulating and Generalizing Plans from Past Experience. In Michalski, R., Carbonell, J., and Mitchell, T., editors, *Machine Learning*, Symbolic Computation, pages 137–161. Springer Berlin Heidelberg.

[Chang and Lin, 2011] Chang, C.-C. and Lin, C.-J. (2011). LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27.

[Chang, 2003] Chang, C.-I. (2003). *Hyperspectral Imaging: Techniques for Spectral Detection and Classification*. Springer.

[Collins et al., 2010] Collins, C. E., Airey, D. C., Young, N. A., Leitch, D. B., and Kaas, J. H. (2010). Neuron densities vary across and within cortical areas in primates. *Proceedings of the National Academy of Sciences*, 107(36):15927–15932.

[Cortes and Vapnik, 1995] Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3):273–297.

[Cover and Hart, 1967] Cover, T. and Hart, P. (1967). Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on*, 13(1):21–27.

[Darwin, 1859] Darwin, C. (1859). On the origins of species by means of natural selection. *London: Murray*.

[DiCarlo and Cox, 2007] DiCarlo, J. J. and Cox, D. D. (2007). Untangling invariant object recognition. *Trends in cognitive sciences*, 11(8):333–341.

[DiCarlo et al., 2012] DiCarlo, J. J., Zoccolan, D., and Rust, N. C. (2012). How does the brain solve visual object recognition? *Neuron*, 73(3):415–434.

[Dodge, 2006] Dodge, Y. (2006). *The Oxford Dictionary of Statistical Terms*. Oxford University Press.

[Donoho and Grimes, 2003] Donoho, D. L. and Grimes, C. (2003). Hessian eigenmaps: Locally linear embedding techniques for high-dimensional data. *Proceedings of the National Academy of Sciences*, 100(10):5591–5596.

[Dorigo et al., 2006] Dorigo, M., Birattari, M., and Stutzle, T. (2006). Ant colony optimization. *Computational Intelligence Magazine, IEEE*, 1(4):28–39.

[Efron, 1982] Efron, B. (1982). *The jackknife, the bootstrap and other resampling plans*, volume 38. SIAM.

[Eiben and Smith, 2003] Eiben, A. E. and Smith, J. E. (2003). *Introduction to evolutionary computing*. Springer.

[Fawcett, 2006] Fawcett, T. (2006). An introduction to ROC analysis. *Pattern recognition letters*, 27(8):861–874.

[Fei-Fei et al., 2007] Fei-Fei, L., Fergus, R., and Perona, P. (2007). Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. *Computer Vision and Image Understanding*, 106(1):59–70.

[Fei-Fei and Perona, 2005] Fei-Fei, L. and Perona, P. (2005). A Bayesian hierarchical model for learning natural scene categories. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 2, pages 524–531 vol. 2.

[Feurer et al., 2015a] Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., and Hutter, F. (2015a). Efficient and Robust Automated Machine Learning. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28*, pages 2962–2970. Curran Associates, Inc.

[Feurer et al., 2015b] Feurer, M., Klein, A., Eggensperger, K., Springenberg, J. T., M., B., and Hutter, F. (2015b). Methods for Improving Bayesian Optimization for AutoML. In *International Conference on Machine Learning (ICML), AutoML 2015 Workshop*.

[Field, 2013] Field, A. (2013). *Discovering Statistics using IBM SPSS Statistics (4th Edition)*. SAGE Publications Ltd.

[Fisher, 1936] Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188.

[Fogel et al., 1966] Fogel, L. J., Owens, A. J., and Walsh, M. J. (1966). *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons.

[Fröhlich et al., 2003] Fröhlich, H., Chapelle, O., and Schölkopf, B. (2003). Feature selection for support vector machines by means of genetic algorithm. In *Tools with Artificial Intelligence, 2003. Proceedings. 15th IEEE International Conference on*, pages 142–148.

[Fukumizu et al., 2004] Fukumizu, K., Bach, F. R., and Jordan, M. I. (2004). Dimensionality Reduction for Supervised Learning with Reproducing Kernel Hilbert Spaces. *J. Mach. Learn. Res.*, 5:73–99.

[Fürnkranz, 1999] Fürnkranz, J. (1999). Separate-and-Conquer Rule Learning. *Artificial Intelligence Review*, 13(1):3–54.

[Genuer et al., 2010] Genuer, R., Poggi, J.-M., and Tuleau-Malot, C. (2010). Variable selection using random forests. *Pattern Recognition Letters*, 31(14):2225 – 2236.

[Glasmachers et al., 2010] Glasmachers, T., Schaul, T., Sun, Y., Wierstra, D., and Schmidhuber, J. (2010). Exponential Natural Evolution Strategies. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*.

[Globerson and Roweis, 2005] Globerson, A. and Roweis, S. T. (2005). Metric learning by collapsing classes. In *Advances in neural information processing systems*, pages 451–458.

[Glover, 1986] Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers & operations research*, 13(5):533–549.

[Goldberg, 1989] Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, Massachusetts.

[Goldberger et al., 2004] Goldberger, J., Roweis, S., Hinton, G., and Salakhutdinov, R. (2004). Neighbourhood components analysis. In *Advances in Neural Information Processing Systems 17*, pages 513–520. MIT Press.

[Gonzalez and Woods, 2002] Gonzalez, R. C. and Woods, R. E. (2002). *Digital Image Processing (2nd Edition)*. Prentice Hall.

[Gordon and Desjardins, 1995] Gordon, D. F. and Desjardins, M. (1995). Evaluation and selection of biases in machine learning. *Machine Learning*, 20(1-2):5–22.

[Gross, 2002] Gross, C. G. (2002). Genealogy of the "grandmother cell". *The Neuroscientist*, 8(5):512–518.

[Guyon and Elisseeff, 2003] Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 3:1157–1182.

[Hall et al., 2009] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18.

[Hamadi et al., 2011] Hamadi, Y., Monfroy, E., and Saubion, F., editors (2011). *Autonomous Search*. Springer Berlin Heidelberg.

[Hansen, 2006] Hansen, N. (2006). The CMA Evolution Strategy: A Comparing Review. In Lozano, J., Larrañaga, P., Inza, I., and Bengoetxea, E., editors, *Towards a New Evolutionary Computation*, volume 192 of *Studies in Fuzziness and Soft Computing*, pages 75–102. Springer Berlin Heidelberg.

[He et al., 2005] He, X., Cai, D., Yan, S., and Zhang, H.-J. (2005). Neighborhood preserving embedding. In *Computer Vision (ICCV), 10th IEEE International Conference on*, volume 2, pages 1208–1213.

[Herwig et al., 2013] Herwig, J., Leßmann, S., Bürger, F., and Pauli, J. (2013). Adaptive Anomaly Detection within Near-regular Milling Textures. In Ramponi, G., Lončarić, S., Carini, A., and Egiazarian, K., editors, *Proceedings of ISPA 2013 - 8th International Symposium on Image and Signal Processing and Analysis*, pages 113–118. University of Zagreb, Croatia, and University of Trieste, Italy, and IEEE, New York.

[Hinton and Roweis, 2002] Hinton, G. E. and Roweis, S. T. (2002). Stochastic neighbor embedding. In *Advances in neural information processing systems*, pages 833–840. The MIT Press.

[Hinton and Salakhutdinov, 2006] Hinton, G. E. and Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507.

[Holland, 1975] Holland, J. H. (1975). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* University of Michigan Press.

[Houle et al., 2010] Houle, M. E., Kriegel, H.-P., Kröger, P., Schubert, E., and Zimek, A. (2010). Can Shared-Neighbor Distances Defeat the Curse of Dimensionality? In Gertz, M. and Ludäscher, B., editors, *Scientific and Statistical Database Management*, volume 6187 of *Lecture Notes in Computer Science*, pages 482–500. Springer Berlin Heidelberg.

[Howell, 2006] Howell, D. C. (2006). *Statistical Methods for Psychology.* Wadsworth Publishing.

[Hu, 1962] Hu, M.-K. (1962). Visual pattern recognition by moment invariants. *Information Theory, IRE Transactions on*, 8(2):179–187.

[Huang and Wang, 2006] Huang, C.-L. and Wang, C.-J. (2006). A GA-based feature selection and parameters optimizationfor support vector machines. *Expert Systems with Applications*, 31(2):231 – 240.

[Huang et al., 2006] Huang, G.-B., Zhu, Q.-Y., and Siew, C.-K. (2006). Extreme learning machine: theory and applications. *Neurocomputing*, 70(1):489–501.

[Huang and Chang, 2007] Huang, H.-L. and Chang, F.-L. (2007). ESVM: Evolutionary support vector machine for automatic feature selection and classification of microarray data. *Biosystems*, 90(2):516 – 528.

[Hung et al., 2005] Hung, C. P., Kreiman, G., Poggio, T., and DiCarlo, J. J. (2005). Fast readout of object identity from macaque inferior temporal cortex. *Science*, 310(5749):863–866.

[Hutter et al., 2011] Hutter, F., Hoos, H., and Leyton-Brown, K. (2011). Sequential Model-Based Optimization for General Algorithm Configuration. In Coello, C., editor, *Learning and Intelligent Optimization*, volume 6683 of *Lecture Notes in Computer Science*, pages 507–523. Springer Berlin Heidelberg.

[Hutter et al., 2009] Hutter, F., Hoos, H. H., Leyton-Brown, K., and Stützle, T. (2009). ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36(1):267–306.

[Hutter et al., 2015] Hutter, F., Lücke, J., and Schmidt-Thieme, L. (2015). Beyond Manual Tuning of Hyperparameters. *KI - Künstliche Intelligenz*, 29(4):329–337.

[Hyvärinen, 1999] Hyvärinen, A. (1999). Fast and robust fixed-point algorithms for independent component analysis. *Neural Networks, IEEE Transactions on*, 10(3):626–634.

[Hyvärinen and Oja, 2000] Hyvärinen, A. and Oja, E. (2000). Independent component analysis: algorithms and applications. *Neural networks*, 13(4):411–430.

[Igel et al., 2008] Igel, C., Heidrich-Meisner, V., and Glasmachers, T. (2008). Shark. *Journal of Machine Learning Research*, 9:993–996.

[Jain et al., 2000] Jain, A., Duin, R. P. W., and Mao, J. (2000). Statistical Pattern Recognition: a Review. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(1):4–37.

[Jain and Zongker, 1997] Jain, A. and Zongker, D. (1997). Feature selection: Evaluation, application, and small sample performance. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 19(2):153–158.

[Jain and Chandrasekaran, 1982] Jain, A. K. and Chandrasekaran, B. (1982). Dimensionality and sample size considerations in pattern recognition practice. *Handbook of statistics*, 2:835–855.

[James and Russell, 1995] James, K. and Russell, E. (1995). Particle swarm optimization. In *Proceedings of 1995 IEEE International Conference on Neural Networks*, pages 1942–1948.

[Japkowicz and Shah, 2011] Japkowicz, N. and Shah, M. (2011). *Evaluating Learning Algorithms : A Classification Perspective*. Cambridge University Press, Cambridge New York.

[Johannsen, 1911] Johannsen, W. (1911). The Genotype Conception of Heredity. *The American Naturalist*, 45(531):129–159.

[Juszczak et al., 2002] Juszczak, P., Tax, D., and Duin, R. (2002). Feature scaling in support vector data description. In *Proc. ASCI*, pages 95–102. Citeseer.

[Kim et al., 2005] Kim, H., Howland, P., and Park, H. (2005). Dimension reduction in text classification with support vector machines. In *Journal of Machine Learning Research*, pages 37–53.

[Kirkpatrick et al., 1983] Kirkpatrick, S., Gelatt, C. D., Vecchi, M. P., et al. (1983). Optimization by simmulated annealing. *Science*, 220(4598):671–680.

[Kotsiantis, 2007] Kotsiantis, S. (2007). Supervised Machine Learning: A Review of Classification Techniques. *Informatica*, 31:249–268.

[Koza, 1992] Koza, J. R. (1992). *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press.

[Kramer, 2008] Kramer, O. (2008). *Self-adaptive heuristics for evolutionary computation*, volume 147. Springer.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In Pereira, F., Burges, C., Bottou, L., and Weinberger, K., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

[Kuncheva and Jain, 2000] Kuncheva, L. and Jain, L. (2000). Designing classifier fusion systems by genetic algorithms. *Evolutionary Computation, IEEE Transactions on*, 4(4):327–336.

[Kunttu and Lepistö, 2007] Kunttu, I. and Lepistö, L. (2007). Shape-based retrieval of industrial surface defects using angular radius Fourier descriptor. *IET Image Processing*, 1(2):231–236.

[Labatut and Cherifi, 2011] Labatut, V. and Cherifi, H. (2011). Accuracy Measures for the Comparison of Classifiers. In *ICIT 2011 The 5th International Conference on Information Technology*.

[Lafon and Lee, 2006] Lafon, S. and Lee, A. B. (2006). Diffusion maps and coarse-graining: A unified framework for dimensionality reduction, graph partitioning, and data set parameterization. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 28(9):1393–1403.

[Lawrence, 2005] Lawrence, N. (2005). Probabilistic non-linear principal component analysis with Gaussian process latent variable models. *The Journal of Machine Learning Research*, 6:1783–1816.

[LeCun and Bengio, 1995] LeCun, Y. and Bengio, Y. (1995). Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks, MIT Press, Cambridge, MA, USA*, 3361(10):255–258.

[LeCun and Cortes, 2010] LeCun, Y. and Cortes, C. (2010). MNIST handwritten digit database. *AT&T Labs Online. Available at* `http://yann.lecun.com/exdb/mnist` *[accessed 28.09.2015 12:00]*.

[Lemke et al., 2013] Lemke, C., Budka, M., and Gabrys, B. (2013). Meta-learning: a survey of trends and technologies. *Artificial Intelligence Review*, pages 1–14.

[Lin et al., 2008a] Lin, S.-W., Lee, Z.-J., Chen, S.-C., and Tseng, T.-Y. (2008a). Parameter determination of support vector machine and feature selection using simulated annealing approach. *Applied Soft Computing*, 8(4):1505 – 1512.

[Lin et al., 2008b] Lin, S.-W., Ying, K.-C., Chen, S.-C., and Lee, Z.-J. (2008b). Particle swarm optimization for parameter determination and feature selection of support vector machines. *Expert Systems with Applications*, 35(4):1817 – 1824.

[Lin and Zha, 2008] Lin, T. and Zha, H. (2008). Riemannian Manifold Learning. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(5):796–809.

[Lowe, 2004] Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110.

[Ma and Fu, 2011] Ma, Y. and Fu, Y. (2011). *Manifold Learning Theory and Applications*. CRC Press.

[Maron and Moore, 1994] Maron, O. and Moore, A. (1994). Hoeffding Races: Accelerating Model Selection Search for Classification and Function Approximation. In Jack D. Cowan, G. T. and Alspector, J., editors, *Advances in Neural Information Processing Systems*, volume 6, pages 59–66, 340 Pine Street, 6th Fl., San Francisco, CA 94104. Morgan Kaufmann.

[McLachlan and Peel, 2000] McLachlan, G. and Peel, D. (2000). *Finite Mixture Models*. Wiley-Interscience.

[Melgani and Bruzzone, 2004] Melgani, F. and Bruzzone, L. (2004). Classification of hyperspectral remote sensing images with support vector machines. *Geoscience and Remote Sensing, IEEE Transactions on*, 42(8):1778–1790.

[Miettinen, 1999] Miettinen, K. (1999). *Nonlinear multiobjective optimization*, volume 12. Springer Science & Business Media.

[Mika et al., 1999] Mika, S., Rätsch, G., Weston, J., Schölkopf, B., and Müller, K. (1999). Fisher discriminant analysis with kernels. In *Neural Networks for Signal Processing IX, 1999. Proceedings of the 1999 IEEE Signal Processing Society Workshop.*, pages 41–48. IEEE.

[Müller, 2012] Müller, M. (2012). *Ein Entwurfsmuster für die multikriterielle Parameteradaption mit Evolutionsstrategien in der Bildverarbeitung*. VDI-Verlag, Fortschritt-Berichte VDI, Reihe 10, Nr. 821.

[Murthy, 1998] Murthy, S. K. (1998). Automatic construction of decision trees from data: A multi-disciplinary survey. *Data mining and knowledge discovery*, 2(4):345–389.

[Niyogi, 2004] Niyogi, X. (2004). Locality preserving projections. In *Neural information processing systems*, volume 16, page 153.

[Ojala et al., 1994] Ojala, T., Pietikainen, M., and Harwood, D. (1994). Performance evaluation of texture measures with classification based on Kullback discrimination of distributions. In *Pattern Recognition, 1994. Vol. 1 - Conference A: Computer Vision amp; Image Processing., Proceedings of the 12th IAPR International Conference on*, volume 1, pages 582–585 vol.1.

[Ojala et al., 2002] Ojala, T., Pietikainen, M., and Maenpaa, T. (2002). Multiresolution gray-scale and rotation invariant texture classification with local binary patterns. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(7):971–987.

[Pan and Yang, 2010] Pan, S. J. and Yang, Q. (2010). A survey on transfer learning. *Knowledge and Data Engineering, IEEE Transactions on*, 22(10):1345–1359.

[Pareto, 1964] Pareto, V. (1964). *Cours d'économie politique*. Librairie Droz (originally published 1896).

[Pearson, 1901] Pearson, K. (1901). On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572.

[Pedregosa et al., 2011] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

[Polikar, 2006] Polikar, R. (2006). Ensemble based systems in decision making. *Circuits and Systems Magazine, IEEE*, 6(3):21–45.

[Pudil et al., 1994] Pudil, P., Novovičová, J., and Kittler, J. (1994). Floating search methods in feature selection. *Pattern Recognition Letters*, 15(11):1119 – 1125.

[Ranawana and Palade, 2006] Ranawana, R. and Palade, V. (2006). Multi-classifier systems: Review and a roadmap for developers. *International Journal of Hybrid Intelligent Systems*, 3(1):35–61.

[Raudys and Jain, 1991] Raudys, S. J. and Jain, A. K. (1991). Small sample size effects in statistical pattern recognition: Recommendations for practitioners. *IEEE Transactions on pattern analysis and machine intelligence*, 13(3):252–264.

[Rechenberg, 1965] Rechenberg, I. (1965). Cybernetic solution path of an experimental problem. *Royal Aircraft Establishment Translation No. 1122, B. F. Toms, Trans.*

[Rechenberg, 1973] Rechenberg, I. (1973). *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog.

[Reif et al., 2012] Reif, M., Shafait, F., Goldstein, M., Breuel, T., and Dengel, A. (2012). Automatic classifier selection for non-experts. *Pattern Analysis and Applications*, pages 1–14.

[Rokach, 2009] Rokach, L. (2009). Taxonomy for characterizing ensemble methods in classification tasks: A review and annotated bibliography. *Computational Statistics & Data Analysis*, 53(12):4046–4072.

[Rosenblatt, 1962] Rosenblatt, F. (1962). *Principles of neurodynamics*. Spartan Book, New York.

[Roweis and Saul, 2000] Roweis, S. T. and Saul, L. K. (2000). Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326.

[Rutter, 2010] Rutter, M. (2010). Gene–environment interplay. *Depression and Anxiety*, 27(1):1–4.

[Sammon, 1969] Sammon, J. W. (1969). A nonlinear mapping for data structure analysis. *IEEE Transactions on computers*, 18(5):401–409.

[Schapire, 1990] Schapire, R. E. (1990). The strength of weak learnability. *Machine learning*, 5(2):197–227.

[Schölkopf et al., 1998] Schölkopf, B., Smola, A., and Müller, K.-R. (1998). Nonlinear component analysis as a kernel eigenvalue problem. *Neural computation*, 10(5):1299–1319.

[Schwefel, 1977] Schwefel, H.-P. (1977). *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie: mit einer vergleichenden Einführung in die Hill-Climbing-und Zufallsstrategie*. Birkhäuser.

[Shaw and Jebara, 2009] Shaw, B. and Jebara, T. (2009). Structure preserving embedding. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 937–944. ACM.

[Silva and Tenenbaum, 2002] Silva, V. D. and Tenenbaum, J. B. (2002). Global versus local methods in nonlinear dimensionality reduction. In *Advances in neural information processing systems*, pages 705–712.

[Sima and Dougherty, 2008] Sima, C. and Dougherty, E. R. (2008). The peaking phenomenon in the presence of feature-selection. *Pattern Recognition Letters*, 29(11):1667–1674.

[Simon, 2003] Simon, R. M. (2003). *Design and analysis of DNA microarray investigations*. Springer Science & Business Media.

[Snoek et al., 2012] Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959.

[Somorjai et al., 2004] Somorjai, R. L., Alexander, M. E., Baumgartner, R., Booth, S., Bowman, C., Demko, A., Dolenko, B., Mandelzweig, M., Nikulin, A. E., Pizzi, N., Pranckeviciene, E., Summers, A. R., and Zhilkin, P. (2004). A Data-Driven, Flexible Machine Learning Strategy for the Classification of Biomedical Data. In Dubitzky, W. and Azuaje, F., editors, *Artificial Intelligence Methods And Tools For Systems Biology*, volume 5 of *Computational Biology*, pages 67–85. Springer Netherlands.

[Sonnenburg et al., 2010] Sonnenburg, S., Rätsch, G., Henschel, S., Widmer, C., Behr, J., Zien, A., Bona, F. d., Binder, A., Gehl, C., and Franc, V. (2010). The SHOGUN machine learning toolbox. *The Journal of Machine Learning Research*, 99:1799–1802.

[Spearman, 1904] Spearman, C. (1904). "General Intelligence," Objectively Determined and Measured. *The American Journal of Psychology*, 15(2):201–292.

[Student, 1908] Student (1908). The probable error of a mean. *Biometrika*, pages 1–25.

[Syswerda, 1989] Syswerda, G. (1989). *Uniform crossover in genetic algorithms*. Morgan Kaufmann Publishers, Inc.

[Teh and Roweis, 2002] Teh, Y. W. and Roweis, S. T. (2002). Automatic alignment of local representations. In *Advances in neural information processing systems*, pages 841–848. The MIT Press.

[Tenenbaum et al., 2000] Tenenbaum, J. B., De Silva, V., and Langford, J. C. (2000). A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323.

[The MathWorks, 2014] The MathWorks, I. (2014). MATLAB and Statistics Toolbox. Software package, Natick, Massachusetts, United States.

[Thornton et al., 2013] Thornton, C., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2013). Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *Proc. of KDD-2013*, pages 847–855.

[Torgerson, 1952] Torgerson, W. S. (1952). Multidimensional scaling: I. Theory and method. *Psychometrika*, 17(4):401–419.

[Van der Maaten, 2009] Van der Maaten, L. (2009). Learning a parametric embedding by preserving local structure. In *International Conference on Artificial Intelligence and Statistics*, pages 384–391.

[Van der Maaten, 2014] Van der Maaten, L. (2014). *Matlab Toolbox for Dimensionality Reduction, http://lvdmaaten.github.io/drtoolbox/ [accessed 28.09.2015 12:00].*

[Van der Maaten and Hinton, 2008] Van der Maaten, L. and Hinton, G. (2008). Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9(2579-2605):85.

[Van der Maaten et al., 2009] Van der Maaten, L., Postma, E., and Van Den Herik, H. (2009). Dimensionality reduction: A comparative review. Technical report, Tilburg University Technical Report, TiCC-TR 2009-005.

[Verbeek, 2006] Verbeek, J. (2006). Learning nonlinear image manifolds by global alignment of local linear models. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 28(8):1236–1250.

[Vert et al., 2004] Vert, J.-P., Tsuda, K., and Schölkopf, B. (2004). A primer on kernel methods. *Kernel Methods in Computational Biology*, pages 35–70.

[Weinberger and Saul, 2009] Weinberger, K. Q. and Saul, L. K. (2009). Distance Metric Learning for Large Margin Nearest Neighbor Classification. *Journal of Machine Learning Research*, 10:207–244.

[Welch, 1947] Welch, B. L. (1947). The generalization of Student's problem when several different population variances are involved. *Biometrika*, pages 28–35.

[Wierstra et al., 2014] Wierstra, D., Schaul, T., Glasmachers, T., Sun, Y., Peters, J., and Schmidhuber, J. (2014). Natural Evolution Strategies. *Journal of Machine Learning Research*, 15:949–980.

[Windhorst and Johansson, 1999] Windhorst, U. and Johansson, H., editors (1999). *Modern Techniques in Neuroscience Research (Springer Lab Manuals)*. Springer.

[Wolfe, 1994] Wolfe, J. M. (1994). Guided Search 2.0 A revised model of visual search. *Psychonomic Bulletin & Review*, 1(2):202–238.

[Wolpert, 1992] Wolpert, D. H. (1992). Stacked generalization. *Neural networks*, 5(2):241–259.

[Wolpert, 1996] Wolpert, D. H. (1996). The lack of a priori distinctions between learning algorithms. *Neural computation*, 8(7):1341–1390.

[Wolpert and Macready, 1995] Wolpert, D. H. and Macready, W. G. (1995). No free-lunch theorems for search. Technical report, Working Paper 95-02-010, Santa Fe Institute.

[Yang et al., 2008] Yang, M., Kpalma, K., and Ronsin, J. (2008). A survey of shape feature extraction techniques. *Pattern recognition*, pages 43–90.

[Zhang et al., 2007] Zhang, T., Yang, J., Zhao, D., and Ge, X. (2007). Linear local tangent space alignment and application to face recognition. *Neurocomputing*, 70(7):1547–1553.

[Zhang and Zha, 2004] Zhang, Z.-y. and Zha, H.-y. (2004). Principal manifolds and nonlinear dimensionality reduction via tangent space alignment. *SIAM Journal of Scientific Computing*, 26(1):313–338.

[Zhou, 2012] Zhou, Z.-H. (2012). *Ensemble methods: foundations and algorithms*. CRC Press.

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Nomenclature

$\cup$	union operator of sets
$\subset$	subset
$\subseteq$	subset or equal set
$\setminus$	relative complement or set-theoretic difference
$\neg$	Boolean negation operator
$\wedge$	logical *and* operator
$\vee$	logical *or* operator
(!)	statistically significant difference according to Welch test
α_{Welch}	level of significance for the Welch test
α	type I error probability
γ_{Gauss}	Gaussian kernel size parameter
Δ_t	generation offset for termination criterion
$\delta_{TargetDimPerc}$	target dimensionality percentage
$\delta_{TargetDimPerc,Max}$	maximum target dimensionality percentage
$\Delta_{fit,max}$	fitness threshold of the multi-pipeline classifier
ϵ_{fit}	minimal fitness improvement for termination criterion
η	polynomial degree
$\Theta_{(index)}$	classification pipeline (with optional *index*)
$\theta_{(index)}$	configuration of a classification pipeline (with optional *index*)
κ	evolutionary metaparameter: lifespan of individuals in number of generations

λ evolutionary metaparameter: number of generated individuals for each new generation (offspring)

μ evolutionary metaparameter: number of surviving individuals in a population

μ_{init} evolutionary metaparameter: size of the initial population

μm micrometer

ρ evolutionary metaparameter: number of parents for each new individual in the offspring generation

σ scalar standard deviation

$\sigma_{Mut,\mathbb{R}}$ mutation parameter for real-valued numbers

$\sigma_{Mut,exp\mathbb{R}}$ mutation parameter for exponentially scaled, real-valued numbers

$\sigma_{Mut,\mathbb{Z}}$ mutation parameter for integer numbers

Σ covariance matrix

τ_1 evolutionary metaparameter: weight for the global mutation adaptation

τ_2 evolutionary metaparameter: weight for the local mutation adaptation

χ_{Mut} initial mutation ratio of the value domain

$\omega_{(index)}$ true class (with optional $index$)

$\widetilde{\omega}_{(index)}$ predicted class (with optional $index$)

$A_{Class(,index)}$ classifier algorithm (with optional $index$)

$A_{(index)}$ algorithm (with optional $index$)

$A_{PreProc(,index)}$ feature preprocessing algorithm (with optional $index$)

$A_{Trans(,index)}$ feature transform algorithm (with optional $index$)

$\mathbb{B}$ Boolean value

$bestConfigSet(N_{Configs})$ set of the best $N_{Configs}$ configurations

b bias value

cm centimeter

$configGraph(\theta)$ configuration graph of a configuration θ

c_{reg} regularization parameter

$d_{name}(\mathbf{x}^{(a)}, \mathbf{x}^{(b)})$ distance function *name* between two vectors

$d_{\mathcal{M}}(\mathbf{x}^{(a)}, \mathbf{x}^{(b)})$ distance function on a manifold $\mathcal{M}$

$D_{Group,l}$ dimensionality of the lth feature group

D_{in} dimensionality of input feature vectors

$D_{(name)}$ other dimensionality of feature vectors (with optional *name*)

D_{Trans} target dimensionality of a configuration θ

$D_{Trans,Max}$ maximum target dimensionality

$\mathfrak{E}$ set of edges in a graph

$E_{Classifier}$ classifier pipeline element

$E_{FeatSel}$ feature selection pipeline element

$E_{PreProc}$ feature preprocessing pipeline element

E_{Trans} feature transform pipeline element

$\exp(x)$ exponential function e^x

$exp\mathbb{R}$ exponentially scaled, real-valued number

$f_{Classifier}(\cdot)$ classifier function

$f_{\mathfrak{E}}$ weight function of edges in a graph

$featGroup(l)$ lth feature group

$f_{FeatImp}(j)$ feature importance metric of the jth feature

fit_{high} upper fitness bound for roulette wheel selection

$fit(I)$ fitness evaluation function of an individual I

fit_{low} lower fitness bound for roulette wheel selection

$\mathbf{fit}(P_t)$ fitness vector of a population P_t

f_{name} function with description *name*

$f_{ProcessPipeline}(\Theta, \mathbf{x})$ process a vector $\mathbf{x}$ with classification pipeline Θ

$f_{TrainPipeline}(\theta, T_{Train})$ train a pipeline with configuration θ and dataset T_{Train}

$f_{\mathfrak{V}}$ weight function of vertices in a graph

$f_{\mathfrak{V},Label}$ description label for vertices in a graph

$\mathbb{G}$ genotype definition

$\mathbf{g}$ vector of optimization metrics

$g_{est}(A, T_{Train})$ generalization estimator for algorithm A on dataset T_{Train}

$g_{Obj}(\cdot)$ optimization objective function

$g_{Obj,Gen}(\cdot)$ generalization estimation function

$g_{Obj,Hyper}(\cdot)$ hyperparameter optimization objective function

HGM hyperparameter genotype mapping

$\mathfrak{H}^*$ search space

$holisticCV(\cdot)$ holistic cross-validation

$H_{\mathbb{R}}$ hyperparameter information for real-valued variables

$h_{\mathbb{R}}$ real-valued hyperparameter value

$h_{\mathbb{R}}^{max}$ maximum value of real-valued hyperparameter variables

$h_{\mathbb{R}}^{min}$ minimum value of real-valued hyperparameter variables

$H_{\mathbb{S}}$ hyperparameter information for categorical variables

h_S categorical hyperparameter value

$H_{\mathbb{Z}}$ hyperparameter information for integer variables

$h_{\mathbb{Z}}$ integer hyperparameter value

$h_{\mathbb{Z}}^{max}$ maximum value of integer hyperparameter variables

$h_{\mathbb{Z}}^{min}$ minimum value of integer hyperparameter variables

$I_{(index)}$ individual of Evolutionary Algorithm (with optional $index$)

i^* index of nearest neighbor

$K(\cdot)$ kernel function

m meter

$\mathcal{M}$ manifold

$\max(\mathbf{x})$ maximum function of a vector of real-valued numbers

$\text{mean}(\mathbf{x})$ average function of a vector of real-valued numbers

$\text{meanVector}(\mathbf{x})$ vector that contains the mean values of a vector

$\min(\mathbf{x})$ minimum function of a vector of real-valued numbers

min minute

mm millimeter

$MPList$ list of classification pipelines of a multi-pipeline classifier

ms millisecond

$\mathbb{N}$ natural number

$N_{Classes}$ number of classes

$N_{Classifiers}$ number of classifiers

$N_{Configs}$ number of configurations

N_{CV} number of cross-validation rounds

$N_{FeatGroupMax}$ maximum number of feature groups

N_{FN} number of false negatives

N_{FP} number of false positives

$N_{Generations}$ number of generations

$N_{Generations,min}$ minimum number of generations

N_{Genes} number of genes in an individual

N_{Groups} number of feature groups

$N_{Hyp}(A)$ number of hyperparameters of algorithm A

$\binom{n}{k}$ binomial coefficient, n choose k

N_{k_1,k_2} frequencies in confusion matrix at position k_1, k_2

$N_{Metrics}$ number of optimization metrics

$\mathcal{N}(\mu_{mean}, \sigma)$ one-dimensional Gaussian normal probability distribution with mean μ_{mean} and standard deviation σ

$\mathcal{N}(\boldsymbol{\mu}_{mean}, \Sigma)$ multivariate Gaussian normal probability distribution with mean vector $\boldsymbol{\mu}_{mean}$ and covariance matrix Σ

N_{Neigh} number of neighbors

$N_{Neurons}$ number of neurons

N_{Opt} number of optimization variables

N_{Pipes} number of classification pipelines

$N_{Pipes,max}$ maximum number of classification pipelines

$N_{PreProc}$ number of preprocessing methods

N_T number of training instances in dataset T

N_{TN} number of true negatives

N_{TP} number of true positives

N_{Trans} number of feature transforms

$\emptyset$ empty set

$\mathcal{O}(\cdot)$ big O notation for complexity (Landau notation)

$optTrajectory$ optimization trajectory

$p_{Feat,min}$ minimum initialization probability for feature selection variables

$p_{Init,\mathbb{B}}$ initial probability for Boolean variables

$p_{Mut,\mathbb{S}}$ mutation parameter for categorical variables

$p_{Mut,\mathbb{S},init}$ initial mutation parameter for categorical variables

$p_{Mut,\mathbb{B}}$ mutation parameter for Boolean variables

$p_{Mut,\mathbb{B},init}$ initial mutation parameter for Boolean variables

p_{name} probability value with $name$

$\mathcal{P}(S)$ power set of set S

$predictionList$ list of predictions of a multi-pipeline classifier

P_t population at generation index t

p_{Welch} probability value of the Welch test

q_{Acc} accuracy

q_{Err} error rate

q_{FPRate} false positive rate

q_i performance value of the ith cross-validation round

q_{OAcc} overall multiclass accuracy

q_{OErr} overall multiclass error rate

q_{Prec} precision

q_{TPRate} true positive rate

$\mathbb{R}$ real-valued number

$\mathbb{R}^+$ positive real-valued number (zero excluded)

$rand$ random number

$randomItem(S)$ random item out of a set S

$rankedConfigs(l_{rank})$ configuration at fitness rank l_{rank}

$rankedConfigsFit(l_{rank})$ fitness of the l_{rank}th configuration

r_{corr} Pearson correlation coefficient

$R_{Group,l}$ description of the lth feature group

round(x) rounding function of a real-valued number to the nearest integer

s second

$S_{(name)}$ set (with optional $name$)

$|S|$ number of items in set S

$S_{Candidates}$ set of candidate solutions

S_{Cat} base set of categorical hyperparameter variables

$S_{Classes}$ set of classes

$S_{Classifiers}$ portfolio set of classifiers

$S_{FeatGroups}$ set of feature groups

$S_{FeatSubSet}$ feature subset of a configuration θ

$\text{sgn}(x)$ sign function

$S_{\mathbf{h}(,index)}$ set of optimization variables (with optional $index$)

$S_{\mathbb{H}}$ set of hyperparameters

$S_{\mathbb{H},A}$ set of hyperparameters of algorithm A

$S_{PreProc}$ portfolio set of preprocessing methods

$S_{\mathbb{S}}$ base set of categorical variables (genetic variables)

$\text{std}(\mathbf{x})$ standard deviation of a vector of real-valued numbers (unbiased estimator)

$S_{\theta}(T_{Train})$ set of all possible configurations defined by dataset T_{Train}

S_{Trans} portfolio set of feature transforms

t generation index of an Evolutionary Algorithm

T_{GT} full ground truth dataset

T_{Train} training dataset

T_{Test} test dataset

$T_{CV,i}$ ith cross-validation dataset division

$T_{CV,Train,i}$ ith cross-validation training dataset

$T_{CV,Valid,i}$ ith cross-validation validation dataset

$\mathcal{U}_{\mathbb{R}}(a,b)$ continuous uniform probability function in interval $[a,b] \subset \mathbb{R}$

$\mathcal{U}_{\mathbb{Z}}(a,b)$ discrete uniform probability function in interval $[a,b] \subset \mathbb{Z}$

$\mathfrak{V}$ set of vertices in a graph

$\mathfrak{v}$ vertex in a graph

$V_{\mathbb{B}}$ genetic variable definition of Booleans

$v_{\mathbb{B}}$ value of Booleans (genetic variables)

$V_{exp\mathbb{R}}$ genetic variable definition of exponentially scaled, real-valued numbers

$v_{exp\mathbb{R}}$ value of exponentially scaled, real-valued numbers (genetic variables)

$v_{exp\mathbb{R}}^{max}$ maximum value of exponentially scaled, real-valued numbers (genetic variables)

$v_{exp\mathbb{R}}^{min}$ minimum value of exponentially scaled, real-valued numbers (genetic variables)

$V_{\mathbb{R}}$ genetic variable definition of linearly scaled, real-valued numbers

$v_{\mathbb{R}}$ value of linearly scaled, real-valued numbers (genetic variables)

$v_{\mathbb{R}}^{max}$ maximum value of linearly scaled, real-valued numbers (genetic variables)

$v_{\mathbb{R}}^{min}$ minimum value of linearly scaled, real-valued numbers (genetic variables)

$V_{\mathbb{S}}$ genetic variable definition of categorical variables

$v_{\mathbb{S}}$ value of categorical variables (genetic variables)

v_{SFR} sample to feature ratio value

$V_{\mathbb{Z}}$ genetic variable definition of integer numbers

$v_{\mathbb{Z}}$ value of integer numbers (genetic variables)

$v_{\mathbb{Z}}^{max}$ maximum value of integer numbers (genetic variables)

$v_{\mathbb{Z}}^{min}$ minimum value of integer numbers (genetic variables)

V_{*} genetic variable of any type

$\mathbf{W}_{(name)}$ linear projection matrix (with optional *name*)

$\mathbf{w}$ weight vector

x scalar value (generally non-bold variables)

$\mathbf{x}$ vector (generally bold and non-capital variables)

x_j jth value of a vector

$\|\mathbf{x}\|_2$ Euclidean length of vector or L_2 norm

$\overline{x}$ mean value

$\mathbf{X}$ matrix (generally bold and capital variables)

$\mathbf{X}^{\top}$ matrix transpose

$\mathbf{X}^{+}$ Moore-Penrose pseudoinverse of a matrix

$\mathbf{x}_{(name)}^{(i)}$ feature vector of the ith instance (with optional *name*)

$\hat{\mathbf{x}}^{(i)}_{(name)}$ transformed feature vector of the ith instance (with optional *name*)

$x^{(i)}_{(name,)j}$ jth element of a vector of the ith instance (with optional *name*)

$x!$ factorial of $x \in \mathbb{Z}$

$\mathbf{x}_{Group,l}$ feature vector of the lth feature group

$y^{(i)}$ class label (optionally of the ith instance)

$\mathbb{Z}$ integer number